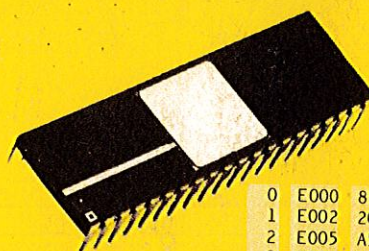


|    |      |        |     |        |
|----|------|--------|-----|--------|
| 0  | ED84 | AD00DD | LDA | \$DD00 |
| 1  | ED87 | 29DF   | AND | #\$DF  |
| 2  | ED89 | 0910   | ORA | #\$10  |
| 3  | ED8B | 8D00DD | STA | \$DD00 |
| 4  | ED8E | C6A5   | DEC | \$A5   |
| 5  | ED90 | D0D4   | BNE | \$ED66 |
| 6  | ED92 | A904   | LDA | #\$04  |
| 7  | ED94 | 8D07DC | STA | \$DC07 |
| 8  | ED97 | A919   | LDA | #\$19  |
| 9  | ED99 | 8D0FDC | STA | \$DC0F |
| 10 | ED9C | AD0DDC | LDA | \$DC0D |
| 11 | ED9F | AD0DDC | LDA | \$DC0D |
| 12 | EDA2 | 2902   | AND | #\$02  |
| 13 | EDA4 |        |     |        |
| 14 | EDA6 |        |     |        |
| 15 | EDA9 |        |     |        |
| 16 | EDAB |        |     |        |
| 17 | EDAC |        |     |        |
| 18 | EDAD |        |     |        |
| 19 | EDAF |        |     |        |
| 20 | EDB2 |        |     |        |
| 21 | EDB5 |        |     |        |
| 22 | EDB6 |        |     |        |
| 23 | EDB7 |        |     |        |
| 24 | EDB9 |        |     |        |
| 25 | EDBB |        |     |        |
| 26 | EDBE |        |     |        |
| 27 | EDC1 |        |     |        |
| 28 | EDC3 |        |     |        |
| 29 | EDC6 |        |     |        |
| 30 | EDC7 |        |     |        |
| 31 | EDC9 |        |     |        |
| 32 | EDCC |        |     |        |
| 33 | EDCD | 20A0EE | JSR | \$EEA0 |
| 34 | EDD0 | 20BEED | JSR | \$EDBE |
| 35 | EDD3 | 2085EE | JSR | \$EE85 |
| 36 | EDD6 | 20A9EE | JSR | \$EEA9 |
| 37 | EDD9 | 30FB   | BMI | \$EDD6 |
| 38 | EDDB | 58     | CLI |        |
| 39 | EDDC | 60     | RTS |        |
| 40 | EDDD | 2494   | BIT | \$94   |
| 41 | EDDF | 3005   | BMI | \$EDE6 |
| 42 | EDE1 | 38     | SEC |        |
| 43 | EDE2 | 6694   | ROR | \$94   |
| 44 | EDE4 | D005   | BNE | \$DEEB |
| 45 | EDE6 | 48     | PHA |        |
| 46 | EDE7 | 2040ED | JSR | \$ED40 |
| 47 | EDEA | 68     | PLA |        |
| 48 | EDEB | 8595   | STA | \$95   |
| 49 | EDED | 18     | CLC |        |
| 50 | EDEE | 60     | RTS |        |
| 51 | EDEF | 78     | SEI |        |
| 52 | EDF0 | 208EEE | JSR | \$EE8E |
| 53 | EDF3 | AD00DD | LDA | \$DD00 |
| 54 | EDF6 | 0908   | ORA | #\$08  |
| 55 | EDF8 | 8D00DD | STA | \$DD00 |
| 56 | EDFB | A95F   | LDA | #\$5F  |
| 57 | EDFD | 2CA93F | BIT | \$3FA9 |

# CORSO DI ASSEMBLER II PER CBM64



|    |      |        |     |        |
|----|------|--------|-----|--------|
| 0  | E000 | 8556   | STA | \$56   |
| 1  | E002 | 200FBC | JSR | \$BC0F |
| 2  | E005 | A561   | LDA | \$61   |
| 3  | E007 | C988   | CMP | #\$88  |
| 4  | E009 | 9003   | BCC | \$E00E |
| 5  | E00B | 20D4BA | JSR | \$BAD4 |
| 6  | E00E | 20CCBC | JSR | \$BCCC |
| 7  | E011 | A507   | LDA | \$07   |
| 8  | E013 | 18     | CLC |        |
| 9  | E014 | 6981   | ADC | #\$81  |
| 10 | E016 | F0F3   | BEQ | \$E00B |
| 11 | E018 | 38     | SEC |        |
| 12 | E019 | E901   | SBC | #\$01  |
| 13 | E01B | 48     | PHA |        |
| 14 | E01C | A205   | LDX | #\$05  |
| 15 | E01E | B569   | LDA | \$69,X |
| 16 | E020 | B461   | LDY | \$61,X |

## E U M - COMPUTER

|    |                 |                                                                 |
|----|-----------------|-----------------------------------------------------------------|
| 1  |                 | !ESEGUE LA SCANSIONE DI TASTIERA IN ATTESA DI UN TASTO PREMUTO  |
| 10 | *=\$C000        | !LOCAZIONE DI INIZIO PROGRAMMA                                  |
| 20 | LOOP JSR \$FF9F | !SCANSIONE DELLA TASTIERA                                       |
| 30 | JSR \$FFE4      | !PRELEVAMENTO DI UN CARATTERE DAL KEYBOARD BUFFER               |
| 40 | BEQ LOOP        | !SALTA ALLA LOC. INDIRIZATA CON LOOP SE NESSUN TASTO E' PREMUTO |
| 50 | RTS             | !RITORNA DA SUBROUTINE                                          |

# **CORSO DI ASSEMBLER II PER CBM 64**

# INTRODUZIONE

Questo manuale si propone di insegnarvi ad usare il linguaggio di programmazione chiamato ASSEMBLER, le possibili applicazioni e darvi un'idea, non del tutto sommaria delle risorse che un computer puo' mettere a disposizione.

Si propone inoltre uno scopo ambizioso e cioe' insegnare all'utente del CBM64 il metodo di programmazione che potra' essere applicato anche ad altri linguaggi.

Naturalmente nessun volume, per quanto vasto puo' comprendere TUTTO neppure sul linguaggio macchina.

Altri volumi sono stati messi a punto dall'EVM COMPUTERS ed in particolare 3 saranno spesso citati se non per la loro indispensabilita' almeno per la complementarita' con questo manuale:

I SEGRETI DEL 1541

LE PERIFERICHE COMMODORE

IL SISTEMA OPERATIVO DEL CBM 64

Infatti per ottenere veramente il massimo e' necessario, a nostro parere, la conoscenza di determinati principi di funzionamento. Sicuramente per il Sistema Operativo, ma non meno per quanto riguarda determinati fenomeni hardware.

Abbiamo constatato inoltre che e' indispensabile che alla teoria si passi con la massima rapidita' possibile alla pratica.

Per questo motivo il corso e' corredato da una cassetta

contenente un programma ASSEMBLER insieme ad altre Utility e che presuppone l'uso di un computer che nel nostro caso e' il CBM 64.

Nessun manuale infatti, a nostro parere, puo' consentire di programmare fino a quando non si eseguono delle applicazioni pratiche ed e' per questo che e' stato ritenuto necessario dotare il volume del supporto costituito dal linguaggio.

Come ripeteremo anche oltre la lettura e le possibilita' applicative di questo manuale non possono prescindere da una conoscenza anche se non necessariamente approfondita del linguaggio Basic presente sul computer.

Indispensabile appare come abbiamo detto la conoscenza del Sistema Operativo del computer sul quale si opera anche se cio' avverra' come necessario complemento a questo corso.

Al termine di quasi ogni capitolo troverete degli esercizi di cui almeno una soluzione sara' data al termine dello stesso manuale.

Infatti un ALGORITMO informatico ammette di solito piu' di una soluzione e comunque sempre piu' di una metodologia risolutiva. Da qui il nostro consiglio per cercare di risolvere i problemi posti con gli esercizi e possibilmente anche gli esempi in altro modo.

Se al termine di questo corso avrete imparato se non a scrivere un complesso programma almeno delle piccole routines che nel vostro specifico campo di applicazione faciliteranno ANCHE la programmazione in Basic, rendendola piu' veloce e semplice, ci saremo ripagati noi della fatica di scriverlo e voi di leggerlo.

## LEZIONE INTRODUTTIVA

Una delle cose che maggiormente scoraggiano il normale utente del computer ad andare oltre il piu' o meno completo apprendimento del Basic e' il numero delle nuove parole, e piu' in generale dei termini da imparare.

Inoltre molti scrittori, in particolar modo quando trattano del Linguaggio Macchina, partono dal concetto che il potenziale utente abbia una conoscenza di elettronica non superficiale e danno quindi per scontato una serie di concetti e di parole.

Come abbiamo gia' detto e come avremo occasione di ripetere tutto cio' che noi pensiamo invece e' che l'utente che si appresta a leggere questo corso abbia una certa conoscenza del Basic.

Per iniziare parliamo della memoria del computer e dei concetti, fondamentali che ad essa si legano.

Una unita' di memoria, in un computer e' come un circuito elettrico che agisce con un interruttore allo stesso modo di quando si entra in una stanza e si accende la luce.

Noi sappiamo che un interruttore puo' essere girato in un senso o nell' altro, ON o OFF, acceso o spento, cioe' puo' avere due posizioni e che resta in quella posizione fin quando non si esegue una operazione che appunto lo sposti da una posizione ad un' altra.

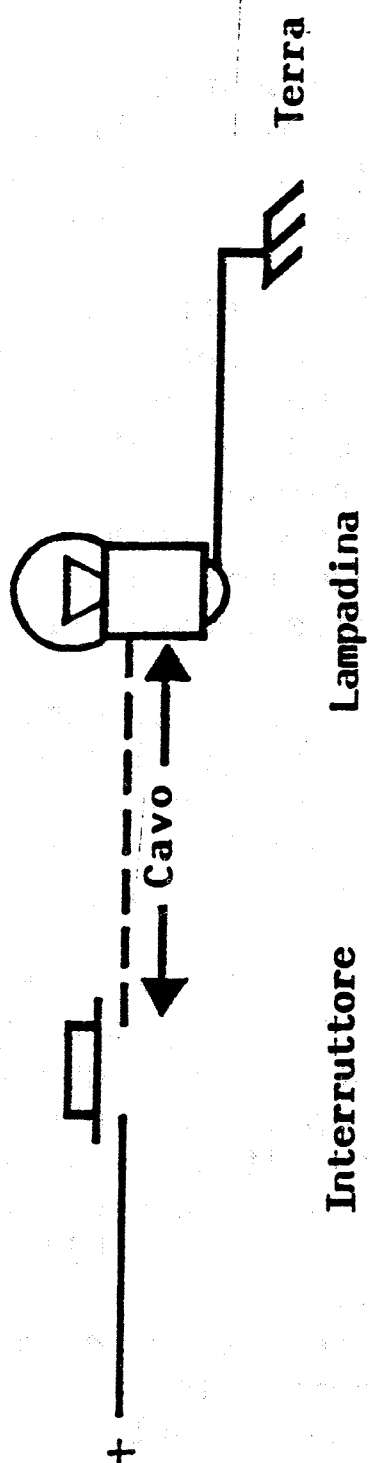
Una unita' di memoria cosi' concepita e che e' l' unita' base del nostro sistema si chiama BIT che e' l' abbreviazione di BINARY DIGIT.

Insistiamo sul concetto di interruttore e di lampadina perche' e' di importanza fondamentale.

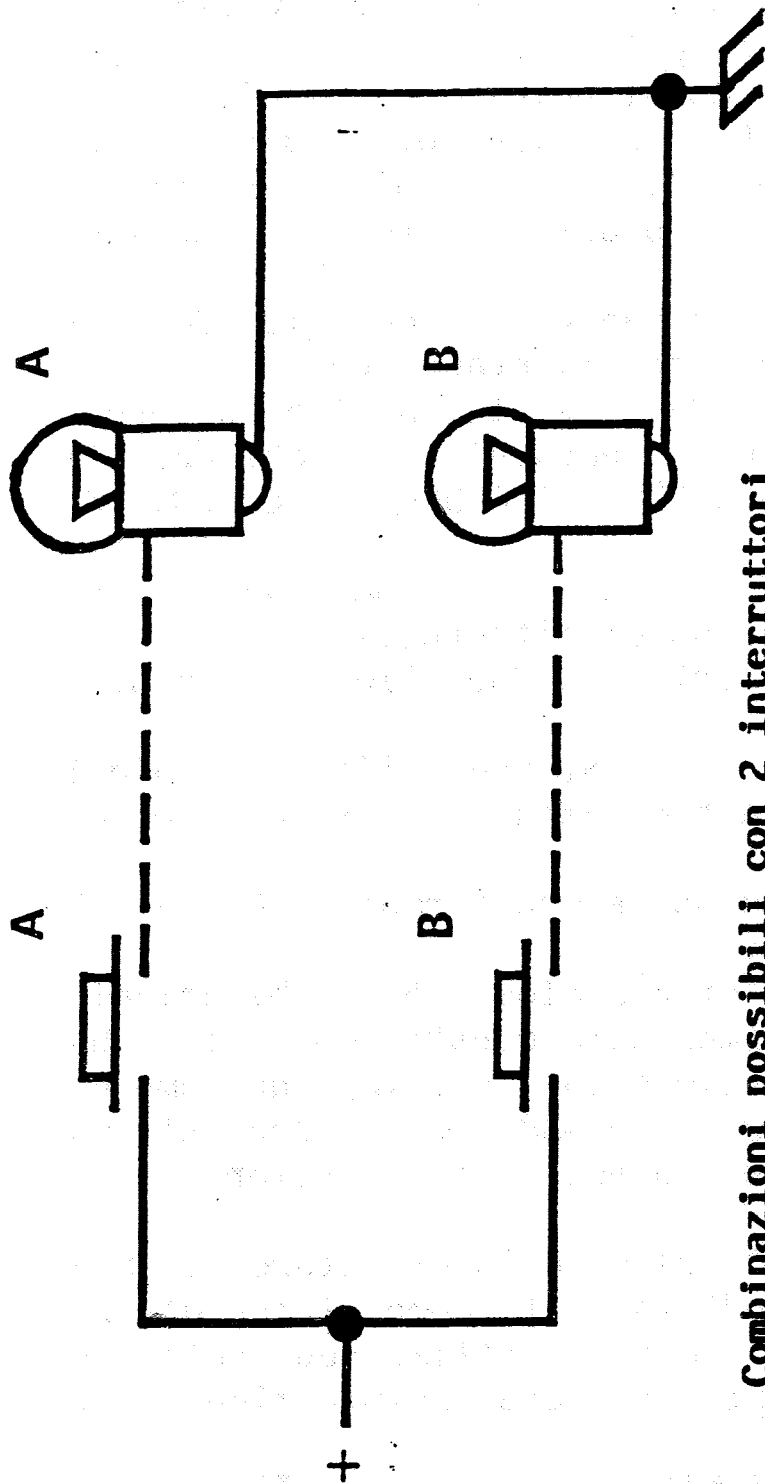
Supponiamo di voler effettuare delle segnalazioni

## CORSO DI ASSEMBLER CBM64

disponendo solo di una lampadina e di un interruttore.



**Circuito con un solo interruttore**



Combinazioni possibili con 2 interruttori

| A   | B   |
|-----|-----|
| off | off |
| off | on  |
| on  | off |
| on  | on  |

Circuito con 2 interruttori

Quando l' interruttore e' ON, in altre parole girato per un certo verso e da ora in poi intendiamo con questo concetto acceso , assumeremo di voler dire di SI.

Al contrario quando e' OFF, cioe' spento, di dire di NO. Possiamo dire che abbiamo due, e due sole, condizioni possibili o STATI della luce o del segnale che vogliamo mandare.

Vediamo ora che cosa puo' accadere quando le luci e quindi gli interruttori da uno diventano due.

Come si puo' chiaramente vedere nella pagina seguente abbiamo quattro differenti possibili combinazioni: a)entrambi OFF; b) A ON e B OFF; c) A OFF e B ON; d) entrambi ON.

Fossiamo ora affermare che essendo 4 combinazioni si possono inviare quattro messaggi differenti.

Con una linea quindi 2 codici, con due linee 4 codici cioe'  $2 \times 1$  e  $2 \times 2$ .

Con tre linee sara'  $2 \times 2 \times 2$  cioe' 8, con 4 linee e quindi quattro interruttori e quattro lampadine  $2 \times 2 \times 2 \times 2$  cioe' 16 e cosi' via.

In altre parole possiamo dedurre una formula che e' 2 elevato al numero delle linee.

Con 8 linee avremo 2 all' ottava cioe' 256 combinazioni.

Dobbiamo ora vedere come impiegare questi concetti. Un sistema particolarmente utile e che noi impiegheremo e' quello del CODICE BINARIO o BINARY CODE che e' un sistema di numerazione che impiega solo due valori 0 e 1.

Possiamo pensare allo ZERO (0) come interruttore spento o OFF e all' UNO (1) o condizione 1 come interruttore acceso o ON. In questo modo, utilizzando solo 8 interruttori potremo impiegare 256 combinazioni di segnali diverse.

Questo raggruppamento di 8 bit e' chiamato BYTE ed e' diventato un' unita' di misura standard che fra l' altro viene impiegata per dare la grandezza della memoria del

computer.

Dopo aver visto i concetti di BIT e i BYTES possiamo affermare che la memoria di un computer altro non e' che una serie di interruttori.

Osserviamo ora che nel computer ci sono due tipi di memorie, una permanente e quindi, per riportarsi all' esempio iniziale, con tutta la serie di interruttori che sono gia' fissati in una determinata posizione e di conseguenza i segnali sono codificati al momento stesso della costruzione della memoria.

Questa si chiama ROM cioe' READ ONLY MEMORY o memoria a sola lettura.

Normalmente questo tipo di memoria, ed e' cosi' nel caso del CBM64, contiene il Sistema Operativo, l' interprete Basic e tutto cio' che consente al computer di comunicare con il mondo esterno o, come meglio si dice di interagire con esso.

Quando scrivete un programma invece il computer lo immagazzina in altra parte della memoria, sempre in forma numerica, e che puo' essere utilizzata piu' volte come lettura e scrittura.

Questo diverso tipo di memoria puo' quindi essere LETTO e SCRITTO e si chiama RAM o RANDOM ACCESS MEMORY.

Gli interruttori posizionati in certo modo o meglio le informazioni scritte in questo tipo di memoria possono come abbiamo gia' detto essere cambiate quante volte si vuole, ma quando si toglie tensione al circuito, o in altre parole si spegne la macchina, queste informazioni vengono irrimediabilmente perse.

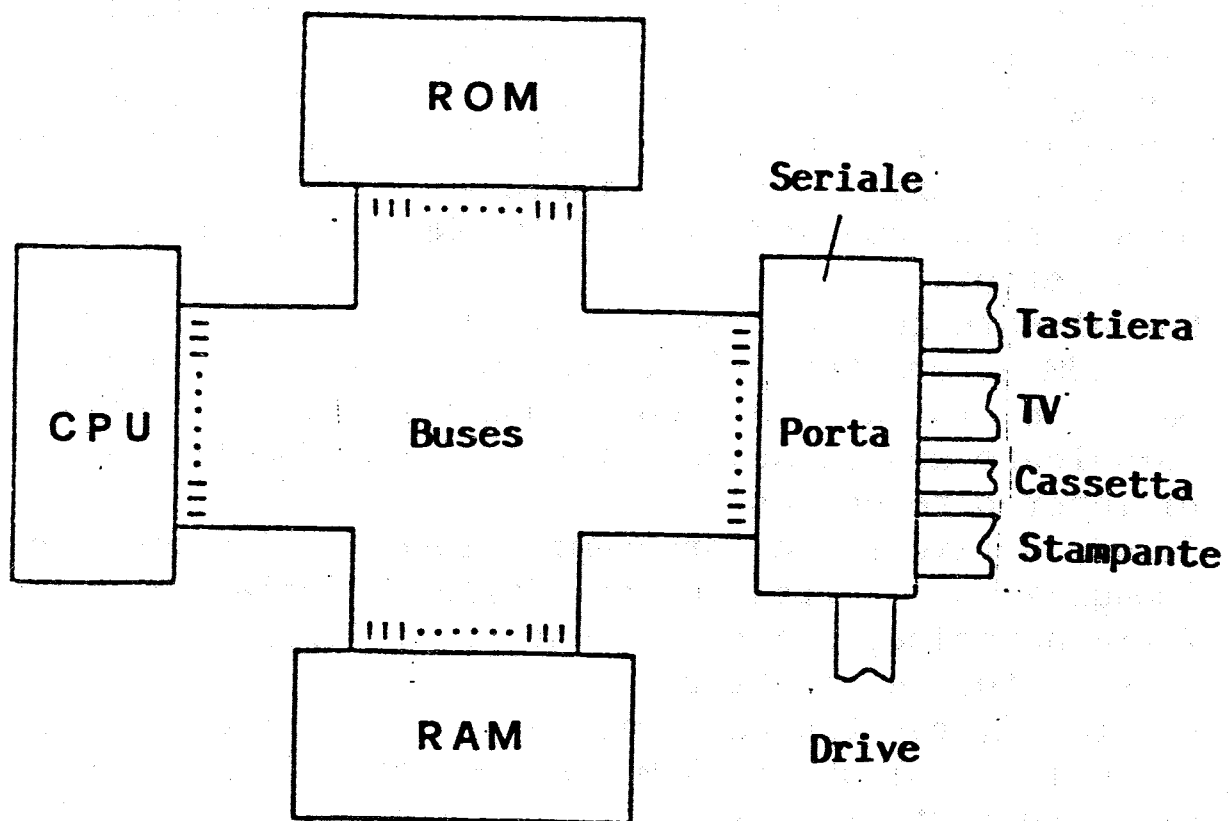


Diagramma a blocchi del CBM 64

## IL MICROPROCESSORE

Nella pagina precedente e' mostrato il diagramma a blocchi del CBM64 o mappa di funzionamento schematica del computer stesso.

I nomi di due componenti del blocco RAM e ROM sono stati chiariti in precedenza. Il blocco con il nome CPU, che sta per CENTRAL PROCESSING UNIT ha una particolare importanza. Infatti questa unita' puo' essere definita come la parte piu' importante del sistema o UNITA' OPERATIVA.

Questa e' costituita da un solo integrato che nel nostro caso e' il microprocessore 6510, cioe' un piccolo pezzo di plastica all' interno del quale e' presente un pezzo di silicio sul quale sono stati riportati con processi di microfotografia una serie di circuiti.

Da questo pezzetto di plastica escono delle connessioni o PIN in numero di 20 per lato.

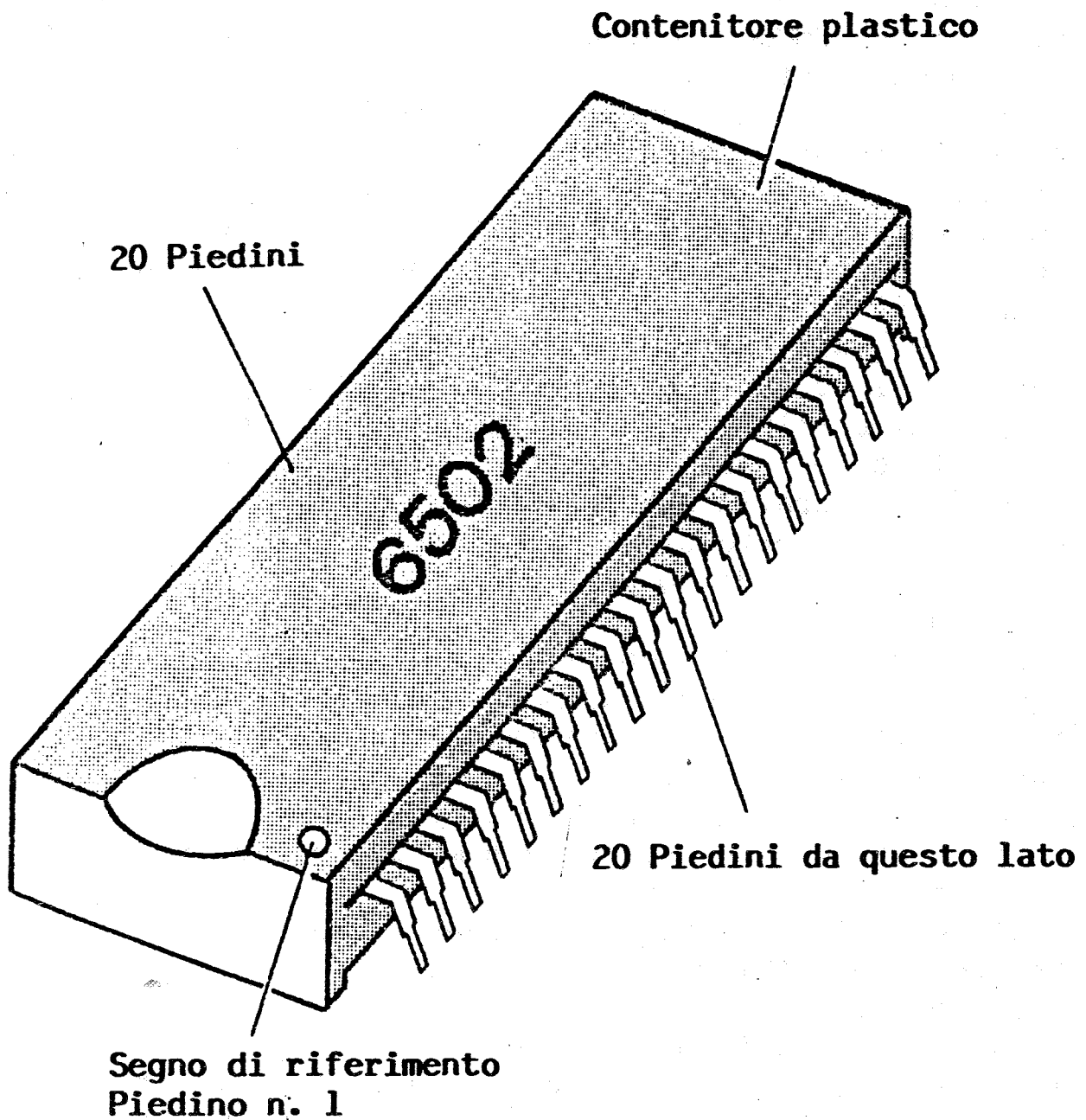
Cosa fa la CPU?

Praticamente tutto ed inoltre le azioni che puo' eseguire sono straordinariamente semplici e poche. E' infatti la combinazione delle stesse azioni che da la capacita' al computer di eseguire calcoli complessi.

La CPU puo' caricare (LOAD) un Byte. In altre parole un Byte in un certo punto della memoria puo' essere immesso all' interno della CPU.

Puo' eseguire anche l' azione opposta cioe' il trasferimento (STORE) in qualsiasi punto, consentito, della memoria.

Queste due azioni sono quelle in cui la CPU impiega la maggior parte del suo tempo operativo e vi renderete conto di questa affermazione continuando nello studio del corso.



Raffigurazione del processore 6502

Per rendersi conto di questa azione in apparenza semplice, vediamo un caso pratico e cioe' cosa accade quando si preme un tasto, ad esempio la lettera G.

La CPU tratta la tastiera come una parte di memoria e il video come se fosse un'altra parte e copia quindi, trasferendo, da una parte all'altra. Naturalmente questo e', come vedremo in seguito,, una semplificazione notevole del concetto ma dovrebbe chiarire l'operazione.

Altre funzioni che esegue la CPU sono quelle aritmetiche spesso solo addizioni e sottrazioni su numeri da 0 a 255.

Quando deve eseguire operazioni piu' complesse come logaritmi, moltiplicazioni, potenze e su numeri piu' grandi vedremo che verranno utilizzate le informazioni, in questo caso, veri pezzi di programma, presenti in ROM.

Oltre questo la CPU esegue anche operazioni LOGICHE che consistono nel confrontare il contenuto dei bits di due Bytes e danno un risultato che dipende dal contenuto dei singoli bits e dall'operazione logica che si esegue.

Un'altra serie di azioni e' quella dei salti o JUMP. Un salto consente un cambio di indirizzo allo stesso modo dell'azione del GOTO nel Basic e quindi da la possibilita' di eseguire dei controlli e di prendere delle decisioni.

Naturalmente tutte queste informazioni viaggiano tramite segnali elettrici in cui sono preventivamente trasformati e, come possiamo osservare dalla tavola del diagramma a blocchi, collegati tramite un BUS o collegamento.

Sotto un altro punto di vista possiamo guardare alla CPU anche come una periferica collegata appunto agli altri componenti del sistema tramite il Bus.

Di identica importanza e' il fatto che lo stesso

microprocessore possa essere programmato inviandogli segnali elettrici.

Questi segnali sono inviati a 8 PINS o piedini chiamati DATA PINS, e per questo non sara' molto difficile comprendere che questi 8 PINS corrispondono agli 8 bits di un Byte.

Ogni Byte della memoria puo' quindi essere connesso con la CPU tramite segnali elettrici.

### I SISTEMI NUMERICI

Come abbiamo gia' detto il microprocessore non comprende altro che il modo di numerazione BINARIO.

Allo stesso modo che il sistema decimale opera sulle cifre 1,2,3,4,5,6,7,8,9,0 il binario opera su 0 e 1.

Per questo si dice che il decimale ed il binario sono dei sistemi di numerazione che operano rispettivamente in base 10 e 2.

Prendiamo per esempio il numero 3783. Questi e' composto da migliaia (3), da centinaia (7), da decine (8) e da unita' (3). Per cui si puo' scrivere:

$$3783 = (3 \times 1000) + (7 \times 100) + (8 \times 10) + (3)$$

o meglio ancora

$$3783 = (3 \uparrow 10) + (7 \uparrow 10) + (8 \uparrow 10) + (3 \uparrow 10)$$

Se noi lo inseriamo in un sistema a base 2 avremo:

$$\begin{aligned} 3873 &= (1 \times 2048) + (1 \times 1024) + (1 \times 512) + (1 \times 256) + (1 \times 128) \\ &+ (1 \times 64) + (0 \times 32) + (0 \times 16) + (0 \times 8) + (1 \times 4) + (1 \times 2) + \\ &(1 \times 1) \end{aligned}$$

oppure con la forma esponenziale come visto in precedenza

$$3873 = (1 \times 2^{11}) + (1 \times 2^{10}) + (1 \times 2^9) + (0 \times 2^8) + (1 \times 2^7) + (1 \times 2^6) + (0 \times 2^5) + (0 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$$

Il numero 3783 in base 2 si scrivera' dunque:

$$3783 = 111011000111$$

Possiamo concludere che il numero detto puo' essere rappresentato in binario con 12 bits.

In generale e' valida la regola che N bits permettono di codificare  $2^N$  numeri che sono compresi fra 0 e  $2^N - 1$ .

I codici operativi del 6510 sono dati in forma di Byte che e' l'insieme di 8 Bits. Percio' possono esistere con il 6510 un massimo di  $2^8 = 256$  codici operativi, anche se in effetti sono di meno.

Per quale motivo e' stato scelto il codice binario dato che noi siamo abituati a contare in base 10?

Il motivo e' nella facilita' di messa in opera dal punto di vista della costruzione del circuito Hardware.

Infatti e' molto piu' semplice realizzare e costruire dei circuiti che possono avere 2 differenti stati, come visto in precedenza (0 o 1, 0 Volts o 5 Volts, SI o NO) invece che realizzare dei circuiti che possono avere 10 differenti stati.

## L' ESADECIMALE

Abbiamo visto il sistema binario, che se e' facile da

realizzare dal punto di vista della circuiteria elettronica non lo e' altrettanto dal punto di vista dell' interazione umana.

E' stato messo a punto quindi un sistema ulteriore di numerazione che vedremo facilitera' molto le operazioni sui Bytes.

Allo stesso modo che il sistema decimale comporta l' impiego di 10 cifre cosi' l' esadecimale ne comporta l' impiego di 16:

0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

di cui mostriamo gli equivalenti:

| ESA | DEC. | BINARIO |
|-----|------|---------|
| 0   | 0    | 0000    |
| 1   | 1    | 0001    |
| 2   | 2    | 0010    |
| 3   | 3    | 0011    |
| 4   | 4    | 0100    |
| 5   | 5    | 0101    |
| 6   | 6    | 0110    |
| 7   | 7    | 0111    |
| 8   | 8    | 1000    |
| 9   | 9    | 1001    |
| A   | 10   | 1010    |
| B   | 11   | 1011    |
| C   | 12   | 1100    |
| D   | 13   | 1101    |
| E   | 14   | 1110    |
| F   | 15   | 1111    |

Vediamo ora come sia possibile convertire un numero

binario in esadecimale e viceversa.

E' molto semplice: basta dividere il numero esadecimale in tante cifre e sostituirle con i 4 valori corrispondenti in binario visti nella tabella precedente.

L' esadecimale 69 sara' quindi:

6 = 0110      9 = 1001

quindi:

\$69 = % 01101001

ed ancora:

\$326D sara'

3 = 0011    2 = 0010    6 = 0110    D = 1101

\$326D = % 0011001001101101

D' altra parte per eseguire l' operazione opposta sara' sufficiente spezzare il numero binario in tanti gruppi di quattro cifre partendo da sinistra.

11111010 sara' FA

## IL CODICE ASCII

Abbiamo visto diversi modi di codifica dei numeri, altri ne vedremo in seguito come il BCD e l' ottale , ma e' ora necessario sapere in che modo codificare i caratteri alfanumerici.

Anche per questo motivo e' stato messo a punto il codice ASCII, acronimo di AMERICAN STANDARD CODE FOR INFORMATION INTERCHANGE.

Si tratta di un codice di 8 bits in cui pero' il bit piu' significativo e' utilizzato per il controllo degli errori. Si tratta del bit di parita'.

L' informazione sara' quindi contenuta in 7 bits cosa che per le regole esposte innanzi permette di codificare 128 caratteri differenti.

Questi 128 codici ci permettono di rappresentare tutte le lettere dell' alfabeto, le cifre oltre ad una serie di caratteri speciali e di controllo.

Li vedremo nella tabella riportata al termine del manuale.

# LEZIONE PRIMA

## IL LINGUAGGIO MACCHINA

Dopo l' introduzione entriamo ora nel vivo del problema Assembler.

Perche' utilizzare il linguaggio macchina?

Quando si acquista un HOME o un PERSONAL computer come il CBM64, normalmente si ha disponibile il linguaggio Basic al momento dell' accensione. Con il Basic si possono risolvere la maggior parte dei problemi abordabili con questo tipo di computer e non e' eccessivamente difficile da adoperare.

Perche' allora imparare un' altro linguaggio? Quali vantaggi da realmente questo nuovo linguaggio di programmazione, che ci auguriamo di poter spiegare bene in questo corso, sul Basic da ripagare i nostri sforzi? Il vostro CBM64 ha inserito, e lo vedete al momento dell' accensione scritto sullo schermo, il linguaggio Basic al suo interno, ma non lo puo' comprendere, almeno direttamente. Infatti quando esegue un comando Basic che magari avete inserito da tastiera, questi, tramite il Sistema Operativo che contiene anche l' interprete Basic, viene convertito in una serie di istruzioni in Linguaggio Macchina, queste eseguibili direttamente dal sistema.

Per eseguire un semplice comando come POKE 1024,10 l' interprete deve fare un mucchio di lavori che portano via tempo. Vediamoli per comprendere meglio.

Per prima cosa l' interprete cerca e controlla che le prime due parole del comando siano presenti nella TAVOLA

## CORSO DI ASSEMBLER CBM64

delle parole riservate. Poi osserva che a questo comando sono necessari 2 argomenti, legge il primo (1024) e lo converte in binario. Ricordiamoci infatti che il computer lavora in binario.

Cerca quindi e trova il secondo argomento (10) e lo converte in binario. Infine scrive questo secondo valore nella locazione di memoria indicata dal primo argomento. Questa serie di operazioni richiede 2 millesimi di secondo. La stessa istruzione potrebbe essere scritta in assembler:

```
LDA #10  
STA 1024
```

L' esecuzione di queste istruzioni richiede 6 milionesimi di secondo che equivale a meno di un trecentesimo del tempo del Basic.

Alcune operazioni come SORT (ordinamenti) e calcoli matematici impiegano realmente molto tempo e se si utilizzano grandi gruppi di dati possono rendersi necessarie anche molte ore con il Basic.

Altre operazioni poi non possono essere eseguite con il Basic come ad esempio la funzione di INTERRUPT.

Un' altro importante vantaggio e' costituito dall' utilizzo della memoria. Infatti un programma in Linguaggio Macchina ben scritto sara' almeno 10 volte piu' corto dell' equivalente in Basic.

Stesso discorso vale per l' occupazione della memoria.

Infatti il Basic richiede 2 Bytes per rappresentare un valore intero compreso fra 0 e 255. In Linguaggio macchina sara' necessario solo 1 Byte.

Tutto questo ci consente di affermare con sicurezza che, anche se presenta una certa difficolta' ad essere appreso, comunque non di molto superiore al Basic, vale veramente la pena di spendere questo tempo.

Facciamo un semplice esempio di somma, aggiungendo il

numero 1 al numero 2.

IN LINGUAGGIO CORRENTE DIREMO :

Aggiungo 1 a 2, quale e' il risultato?

IN BASIC SI POTREBBE IMPOSTARE:

```
10 A=1
20 B=2
30 C= A+B
40 PRINT C
```

IN CODICE MACCHINA (per il 6510):

```
A2 01
8E 84 03
A9 02
6D 84 03
8D 85 03
60
```

Cio' che non risulta molto intelleggibile.  
Diamo di seguito lo stesso problema in ASSEMBLER con un  
breve commento per ogni linea.

```
LDX #1    Carica il registro X con '1'
STX 900    Immagazzina il contenuto di X in 900
```

```
LDA #2   Carica '2' nell' Accumulatore
ADC 900   Aggiungi all' accumulatore il contenuto dell'
          indirizzo di memoria 900
STA 901   Metti il contenuto dell' Accumulatore nella
          locazione di memoria 901.
RTS       Ritorno da Subroutine in L.M.
```

Ecco uno dei motivi per il quale si utilizza l' ASSEMBLER che come possiamo vedere e' molto piu' facile da leggere e da ricordare che non il Linguaggio Macchina visto prima.

La funzione di tutti gli Assembler e' quindi quella di tradurre un programma scritto in codici mnemonici, e quindi semplici da ricordare, in Codice Macchina.

E' possibile anche inserire codici macchina direttamente in memoria e questo sara' visto piu' avanti in altri capitoli di questo libro.

### L' ACCUMULATORE

Il cuore del microprocessore 6510 e' un registro chiamato ACCUMULATORE ed abbreviato con la lettera A.

Si tratta di un registro ad 8 bit attraverso il quale passeremo quasi sempre e che quindi puo' immagazzinare numeri da 0 a 255.

Le istruzioni del 6510 consentono di scrivere direttamente entro questo registro.

Una delle istruzioni appena viste e':

```
LDA #   Load Accumulator using Immediate Mode
```

cioe' carica l'Accumulatore con il valore che segue (modo immediato).

#### NOTA

LDA puo' essere usato nelle due forme:

LDA #n che carica l' Accumulatore con il valore che segue, appunto MODO IMMEDIATO.

LDA seguito dalla locazione di memoria ed in tutte le forme che vedremo in seguito, che invece carica l' Accumulatore con il valore presente in memoria.

Un' altra istruzione vista consente di trasferire un numero immagazzinato nell' Accumulatore in una specificata locazione di memoria. Questa istruzione e':

STA Store contents of Accumulator in the address specified

cioe' immagazzina il contenuto dell' Accumulatore in un dato punto della memoria.

#### NOTA

Nel caso del CBM64 se la locazione di memoria nella quale si copia il contenuto dell' Accumulatore e' compresa fra i valori decimali 1024 e 2023, allora il carattere corrispondente di quel numero sara' visualizzato sullo schermo appunto perche' questa e' l' area riservata alla memoria di schermo.

Il comando STA esegue una vera e propria COPIA dell'

## CORSO DI ASSEMBLER CBM64

Accumulatore, lasciando il contenuto dell' accumulatore stesso inalterato.

Creeremo ora un programma che immettera' un numero nell' Accumulatore e dopo lo visualizzera' nel punto piu' in alto a sinistra dello schermo.

Prima pero' e' bene puntualizzare qualcosa circa l' Assembler che useremo.

Diamo quindi una prima veloce spiegazione di utilizzo del programma ASSEMBLER presente sul lato A della Cassetta.

1) Controllare che la cassetta sia all' inizio cioe' che sia stata riavvolta completamente.

2) Eseguire il caricamento con LOAD.

3) Eseguire il comando RUN.

Particolarita' essenziale di questo assembler e' quella di consentire, pur con tutto il gruppo di istruzioni tipico di questo linguaggio, la possibilità di programmare in maniera molto simile a quella adoperata con il Basic

Viene infatti utilizzato lo stesso sui stema di numerare le linee 10, 20, 30 ecc. con in piu', come spiegheremo fra breve l' opzione di AUTOMATIC LINE NUMBERING.

Da osservare che solo la prima linea del programma, poniamo la 10, dovra' indicare la locazione di memoria da cui dovra' partire il programma assemblato o PROGRAMMA OGGETTO.

Quindi, a grandi linee i programmi saranno scritti con

il seguente metodo

| N. LINEA | LABEL | OPCODE | VALORE |
|----------|-------|--------|--------|
|----------|-------|--------|--------|

Esempio 1:

|    |                       |   |   |
|----|-----------------------|---|---|
| 10 | *=INDIRIZZO DI INIZIO |   |   |
| 20 | ISTRUZIONE O OPCODE   |   |   |
| 30 | ISTRUZIONE O OPCODE   |   |   |
| .  | .                     | . | . |
| .  | .                     | . | . |
| .  | .                     | . | . |

#### NOTA

Quando si inizia a scrivere un programma in Assembler e' necessario conoscere in quale punto della memoria si desidera poi immagazzinare il programma stesso.

Nel CBM 64 ci sono numerosi punti della memoria disponibili per questa funzione di immagazzinamento, tuttavia per il momento e per brevi programmi, possiamo utilizzare il Buffer di cassetta grande 192 Bytes e che va dalla locazione decimale 828 a 1019.

La seconda annotazione sull' Assembler e' che per il momento utilizzeremo solo il formato decimale per i nostri numeri, mentre piu' in avanti vedremo come usare le altre notazioni numeriche disponibili.

#### LE LABEL

Sono quei nomi da mettere prima dei codici mnemonici e che servono all' interno del programma come riferimento

per i vari salti.

Sempre con riferimento al Basic possiamo considerare le LABEL come i nomi delle Subroutines solo che in questo caso, invece di utilizzare un indirizzo si utilizza un nome al quale corrisponde un indirizzo.

Prima di passare a scrivere il nostro primo programma e' opportuno notare che si possono eseguire delle correzioni anche semplicemente riposizionandosi con il cursore. In altre parole utilizzando la funzioni di editing del Basic che rimangono attive.

Dopo aver inserito il programma questi deve essere ASSEMBLATO.

Si deve cioe' far eseguire il comando ASSEMBLE perche' cio' che abbiamo scritto in codice mnemonico (LDA, STA, ecc) possa essere convertito, cioe' tradotto in Codice Macchina che, come abbiamo detto in precedenza, puo' essere direttamente eseguito o elaborato dal computer. Useremo un SYS di chiamata all' indirizzo di partenza. Nel nostro caso quindi SYS 828.

### NOTA

Ricordiamo che tutte e due i metodi devono comunicare al computer di ritornare in ambito Basic ( almeno per il momento ) perche' in caso contrario si entrerebbe in un LOOP o ciclo infinito per uscire dal quale occorrerebbe resettare il sistema intero.

Il comando che esegue questa funzione di ritorno e che quindi deve essere messo al termine di ogni programma Assembler e':

**RTS** ReTurn from Subroutine

cioe' ritorno da Subroutine.

funzionare il nostro programma dovremo eseguire i seguenti passi:

Inserire un ReTurn da Subroutine al Basic con RTS. Questo comando alla fine di ogni programma e' indispensabile perche' in caso contrario si entrerebbe in un LOOP infinito cioe' il sistema continuerebbe a far girare il programma senza sapere dove e quando fermarsi.

Esempio 2:

```
10 *=828
20 LDA $01
30 STA $0400
40 JSR PIPPO
50 RTS
60 PIPPO LDA $02
70 STA $0401
80 RTS
```

#### COMMENTO

Linea 10 E' stato fissato l' indirizzo di partenza del programma assemblato.

Linea 20 L' accumulatore e' stato caricato con il valore contenuto nella locazione 1

Linea 30 Il valore dell' Accumulatore e' posto nella prima locazione di schermo.

Linea 40 Salto ad una Subroutine nel programma stesso.

Linea 50 Ritorno da Subroutine.

Linea 60-80 In questa riga si definisce la Subroutine PIPPO che in questo caso, scrivera' nella seconda locazione di schermo il valore contenuto nell' indirizzo

di memoria \$02.

Una LABEL puo' definire anche un' indirizzo. Vedremo comunque in un apposito capitolo l' impiego di questo potente strumento.

Vediamo ora il resto dei comandi del nostro Assemblatore.

### NOTA

In alcuni computer piu' vecchi puo' non apparire nulla sul video. In questo caso aggiungere le linee:

```
42 LDA #1
43 STA 55296
44 STA 55297
```

che serviranno appunto a dare il colore ai caratteri sullo schermo.

### I COMANDI DELL' ASSEMBLER

Quando il programma e' stato scritto, dovrebbe per prima cosa essere memorizzato allo stesso modo di un programma Basic, su disco o su nastro.

Naturalmente dopo non deve essere dato il RUN perche' il computer non riconoscerebbe un programma valido nel nostro SORGENTE.

Verra' invece dato il comando ASSEMBLE e, se non ci sono errori potra' essere visualizzata una schermata del genere:

```
ASSEMBLE
```

\* PASS (1)\*

\* PASS (2)\*

\* PASS (3)\*

\*\*\*ASSEMBLY COMPLETE\*\*\*\*

START ADDRESS \$0302

END ADDRESS \$0600

In altre parole il comando ASSEMBLE trasforma il programma sorgente in programma OGGETTO con una compilazione in 3 passi.

Se viene rilevato un errore, questi lo sarà nel primo o nel secondo passaggio attraverso il sorgente. Verrà quindi fermata la procedura di ASSEMBLAGGIO e verrà visualizzato l'errore e la linea relativa. Ricordiamo che poi il programma assemblato o codice oggetto potrà essere fatto girare con un SYS all'indirizzo di partenza.

Vediamo ora gli altri comandi che questo linguaggio ci mette a disposizione.

## AUTO

Sintassi: AUTO (linea di inizio), (incremento)

Funzione: Per incrementare e visualizzare numeri di linea di programma automaticamente.

L'uso del comando AUTO senza specificazione dei parametri, cioè della linea di inizio né dell'intervallo fra le linee, visualizzerà automaticamente la linea 100, poi la linea 110, la 120 e così via.

Come si può notare dal formato però questo comando si può adoperare per specificare un indirizzo di partenza

## CORSO DI ASSEMBLER CBM64

diverso da 100 ed un incremento diverso da 10.

Naturalmente si tratta di un comando da dare in modo diretto e che quindi non potrà essere impiegato in modo programma.

Per annullare questa funzione premere RETURN due volte di seguito.

### Esempio

Si intende iniziare un programma, o continuarlo, a partire dalla linea 50 con un incremento di 5 numeri di linea.

Comando AUTO 50,5

Risultato: Sullo schermo sara' visualizzato la prima linea con un 50 ed una volta date le istruzioni ed il RETURN relativo, verra' visualizzato l' inizio della linea successiva che percio' sara' la 55.

## DELETE

Sintassi: DELETE (linea di inizio)-(linea di fine)

Funzione: Cancella linea di programma dalla memoria del CBM64.

Questo comando serve a cancellare linee di programma senza doverlo effettuare con l' Editing. Il suo funzionamento e' simile a quello del comando LIST.

### Esempio

Cancellare le prime due linee del seguente programma.

```
100 LDX $02
```

```
110 PIPPO STX $1000
```

```
120 INX
130 BNE PIPPO
140 RTS
```

.

Comando: DELETE 100-110 oppure DELETE -110.

Risultato: Eseguendo un LIST potremo osservare che rimangono del precedente programma solo le linee a partire dalla 120.

## FIND

Sintassi: FIND(dato), (linea di inizio)-(linea di fine)

Funzione: Ricerca nel programma un qualsiasi dato e visualizza la linea o le linee dove e' presente.

Anche questo comando puo' essere usato sia per cercare in tutto il programma, omettendo i dati di linea, sia per una ricerca parziale assegnando gli opportuni valori di inizio e fine linea di ricerca.

Esempio: Si desidera trovare il carattere \$ nel seguente programma.

```
100 LDX $02
110 PIPPO STX $1000
120 INX
130 BNE PIPPO
140 RTS
```

Comando: FIND \$

Risultato: vengono visualizzate le seguenti linee:

```
100 LDX $02
110 PIPPO-- STX $1000
```

## FORMAT

Sintassi: FORMAT (linea di inizio)-(linea di fine)

Funzione: Visualizza il programma secondo lo schema assembler.

Per una corretta visualizzazione si raccomanda di lasciare uno spazio dopo le LABEL, i comandi mnemonici, ecc. I parametri di linea inizio e fine sono opzionali. Per il resto il comando funziona come il LIST.

## NUMBER

Sintassi: NUMBER (valore)

Funzione: Visualizza sullo schermo i corrispondenti valori esadecimali, decimali, ottali e binari.

Questo comando puo' essere solo utilizzato in forma diretta e non in modo programma. I valori possono essere dati in decimale, esadecimale, facendoli precedere dal simbolo \$, ottali preceduti da @, binari preceduti da %.

Esempio: NUMBER 1024

Risultato:

```
$0400 1024 @002000 %000001000000000000
```

**TABLE**

Sintassi: TABLE

Funzione: Visualizza le labels e le locazioni della memoria in cui si trova il loro opcode o codice operativo dopo pero' che e' stato eseguito il processo di assemblaggio.

**ASSEMBLE**

Sintassi: ASSEMBLE

Assembla il programma cosi' detto SORGENTE e lo trasforma in OGGETTO. In altre parole converte i codici mnemonici in valori riconoscibili dal microprocessore. Se nessun comando di locazione inizio memoria e' presente all' interno del programma viene assunto come inizio la locazione di memoria \$033C.

**DISASSEMBLE**

Sintassi: DISASSEMBLE (indirizzo di inizio) - (indirizzo di fine)

Funzione: Visualizza la zona di memoria compresa fra i due parametri di inizio e di fine, disassemblandone il contenuto.

Esempio: DISASSEMBLE \$A000- \$A030

Risultato: Disassembla il contenuto delle locazioni di memoria da \$A000 a \$A030 compreso.

I parametri che seguono questo comando possono essere

dati sia in forma decimale es:

DISASSEMBLE 40960-41008

sia in forma ottale o binaria preceduti rispettivamente dai segni @ e %.

### NOTA

Eventuali commenti al programma, in altre parole le REM del Basic, devono essere preceduti dal punto esclamativo (!).

Esempio:

20 LDA #1 !carica l' accumulatore con il valore 1.

Oltre quanto abbiamo appena visto esistono anche alcune facilitazioni nell' uso di questo linguaggio che sono date dai simboli:

- > Maggiore
- < Minore
- = Uguale
- + Piu'
- Meno

Questi vengono utilizzati di solito in combinazione con le Labels anche se resta la possibilita' di impiegarli con numeri interi.

I simboli di > e < consentono rispettivamente di utilizzare la parte alta e la parte bassa di un valore sia che questi sia immesso come costante che come Label. Per esempio:

STA >\$0400

Immetterà il valore presente nell' Accumulatore NON nella locazione \$0400, ma nella parte alta di questa locazione, cioè in \$04. Infatti un valore, ed in questo caso ricordiamo che abbiamo fatto un esempio con valori esadecimali, può essere scomposto in due parti, come vedremo meglio anche in seguito, la parte alta, in questo caso \$04, e la parte bassa, sempre in questo caso \$00. Quindi, mentre il segno > fa utilizzare di questo valore SOLO la parte più significativa o PARTE ALTA il segno < fa impiegare la parte BASSA o meno significativa. L' utilizzo con le Labels non è diverso. Supponiamo di aver assegnato precedentemente il valore ad una Label e di voler poi utilizzare solo una parte:

```
10 * = 828
20 BASE = $0360
30 LDA #1
40 STA > BASE
50 ....
60 ....
```

Questo programma esegue il caricamento in Accumulatore del valore 1 e lo scrive poi nella locazione di memoria \$03 invece che in \$0360.

Il segno di uguale, che abbiamo già visto in precedenza, può, nel caso delle Label assegnare a questa un certo valore oppure il valore di un'altra Label.

Intuitive sono le funzioni di + e - che servono per sottrarre o sommare valori dati. Ad esempio:

LDA 1024+5

## CORSO DI ASSEMBLER CBM64

Caricherà in Accumulatore il valore presente nella locazione 1029.

Cio' può rivelarsi molto utile quando ad esempio si ha una tavola posizionata in una certa parte della memoria e si vogliano richiamare i valori con l' utilizzo di una sola Label. Vediamone un esempio:

```
10 *=828
20 BASE BYT 23,34,35
30 LDA BASE
40 LDA BASE+1
50 LDA BASE+2
60.....
70.....
```

Come si può semplicemente constatare con questo sistema si possono immettere in memoria 3 valori e precisamente 23 alla locazione 828, 34 alla locazione 829 e 35 alla locazione 830.

Quindi nei successivi passi e con l' impiego di una sola Label (BASE) che indica l' inizio della tavola si caricherà l' Accumulatore con i contenuti delle locazioni 828 (BASE), 829 (BASE+1), 830 (BASE+2).

Da notare che in questo esempio non è che l' Accumulatore al termine contenga 3 valori ma solo l' ultimo che è stato caricato.

Ricordiamo infatti che ogni volta che si carica un valore nell' Accumulatore si cancella quello precedentemente presente.

## LEZIONE SECONDA

### I REGISTRI INDICE

Oltre all' Accumulatore, il 6510 ha due registri sempre di un solo Byte, detti REGISTRI INDICE:

**REGISTRO INDICE X**

**REGISTRO INDICE Y**

Ognuno di questi che d' ora in poi chiameremo semplicemente registro X o Y ha, come l' Accumulatore, la possibilita' di immagazzinare valori nell' intervallo da 0 a 255 essendo registro da 8 bits.

### NOTA

Ricordiamo ancora una volta che un registro e' una locazione di memoria nella quale puo' essere caricato un valore.

Questo valore di norma e' compreso, come abbiamo detto in un' intervallo fra 0 e 255 per i registri da 8 bits e fra 0 e 65535 per i registri da 16 bits.

Per il momento inoltre i registri X e Y sono mostrati con funzionamento simile fra loro sebbene in effetti differiscano come comportamento come vedremo nel corso del volume.

Il grande vantaggio di questi registri indice e' che il valore in essi contenuto puo' essere incrementato o decrementato, di 1 per volta, in modo semplice.

Naturalmente non sono solo questi i vantaggi e le possibilita' che vedremo oltre.

Altro punto da prendere in considerazione e' l' ALU o ARITHMETIC AND LOGIC UNIT cioe' unita' aritmetico-logica che si trova all' interno del microprocessore stesso ed e' da questi usata appunto per tutte le operazioni aritmetico logiche.

La ALU ha due ingressi per i dati sui quali esegue le operazioni ed un' uscita tramite la quale i risultati delle operazioni stesse vengono inviati all' Accumulatore.

Quindi tutti i dati sui quali opera il 6510 passano almeno una volta attraverso l' Accumulatore. e da questo si intuisce l' importanza di questo Registro.

La strada sulla quale i dati passano si chiama:

### DATA BUS

Mentre nel seguito di questo capitolo e dei prossimi vedremo come i dati passino da un registro all' altro e come si possa aver accesso alla memoria, esaminiamo ora i comandi per mettere in funzione questi due registri.

**LDX** Load index register X

Carica il registro indice X con i dati da un' indirizzo di memoria. Per esempio:

**LDX 900**

Consente di immettere nel registro X il dato presente nella locazione di indirizzo decimale 900.

LDX differisce da LDA# perche', a parte il fatto che un' istruzione carica il registro X e l' altro l' Accumulatore, LDA# e' un comando IMMEDIATO.

In altre parole quando il 6510 trova l' equivalente in

codice assemblato dell' istruzione LDA#, leggerà il valore immediatamente seguente il comando, ( cioè il suo parametro) e lo caricherà nell' Accumulatore.

Con il comando LDX invece il microprocessore prenderà il dato scritto immediatamente dopo l' istruzione e lo considererà come INDIRIZZO a cui saltare per caricare un dato.

Con la seguente istruzione pertanto:

LDX 900

il 6510 eseguirà un salto alla locazione di memoria 900, preleverà il dato ivi presente e lo caricherà nel registro indice X.

#### NOTA

Anche nel caso di questa istruzione ci troviamo di fronte ad una COPIA di un contenuto di memoria perché ciò che era nella locazione di memoria 900 viene copiato nel registro X, ma lo stesso valore resta sempre anche all' indirizzo 900.

Poiché è necessario disporre spesso di questi registri liberi ecco un' istruzione che consente di immagazzinare invece il contenuto di uno di questi registri in una data locazione di memoria. L' istruzione è:

STX STore X in a address

Cioè immagazzina il contenuto del registro X in una data zona di memoria. Per esempio:

STX 1024

Comunica al computer di immettere il contenuto del registro indice X nella locazione di memoria 1024.

## CORSO DI ASSEMBLER CBM64

### PROGRAMMA 3

```

10 *=828
20 LDA #1
30 STA 1024
40 STA 55296
50 LDX 1024
60 STX 1026
70 STA 55298
80 RTS

```

INIZIO 828

```

LDA #1      Carica 1 nell' Accumulatore
STA 1024    Carica il contenuto dell' Accumulatore in
1024.
STA 55296   Immagazzina il contenuto dell' Accumulatore
nella locazione 55296 per visualizzare la
rappresentazione dello schermo in bianco.
LDX 1024    Carica nel registro X il contenuto di 1024
(1).
STX 1026    Immetti il contenuto di X in 1026
STX 55298   Immetti il contenuto di X in 55298 per
visualizzare in BIANCO anche questo carattere.
RTS         Ritorno da subroutine.

```

Dopo aver scritto, e compreso, il listato eseguire l' opzione ASSEMBLE e poi il SYS 828 per far girare il programma.

Se tutto e' stato eseguito correttamente dovrebbero essere visualizzate:

A spazio A

entrambe le lettere in lato a sinistra in bianco.

### NOTA

Le possibili soluzioni dei seguenti esercizi sono date al termine del manuale.

Abbiamo parlato di possibili soluzioni in quanto come abbiamo già detto, per risolvere un dato problema sono possibili più soluzioni algoritmiche.

### Esercizio 1

Caricare l' Accumulatore con 1 visualizzando questo nella locazione 1024 in verde.

### Esercizio 2

Scrivere il vostro nome in alto a sinistra dello schermo.

### Esercizio 3

Scrivere la lettera X in ognuno dei quattro angoli dello schermo.

## IL REGISTRO Y

Dopo aver visto le istruzioni relative al registro X vediamo ora quelle del registro Y in molti casi del tutto eguali.

LDY Load register Y

## CORSO DI ASSEMBLER CBM64

cioe' carica il registro Y con il contenuto di una data locazione di memoria.

Anche per questo registro vale il discorso fatto in precedenza con X.

STY   Store in Y

cioe' immagazzina i dati contenuti nel registro Y in un determinato indirizzo di memoria.

Per molte operazioni, MA NON PER TUTTE, i registri Y e X possono essere intercambiabili. Per questo il programma 3 visto in precedenza puo' essere scritto:

PROGRAMMA 4

```
10 *=828
20 LDA #1
30 STA 1024
40 STA 55296
50 LDX 1024
60 STX 1026
70 STX 55298
80 RTS
```

oppure

PROGRAMMA 4/a

```
10 *=828
20 LDA #1
30 STA 1024
40 STA 55296
50 LDY 1024
```

```
60 STY 1026  
70 STY 55298  
80 RTS
```

La necessita' di girare o passare rapidamente i dati da un registro all' altro anche durante l' esecuzione di un programma evidenzia l' utilita' delle seguenti istruzioni:

**TAX** Transfer Accumulator in X.

Cioe' trasferisci il contenuto dell' Accumulatore nel registro X.

Usando ad esempio questo comando si puo' riscrivere il programma 4 in questo modo:

PROGRAMMA 4/B

```
10 *=828  
20 LDA #1  
30 STA 1024  
40 STA 55296  
50 TAX  
60 STX 1026  
70 STX 55298  
80 RTS
```

Anche in questo caso avremo il risultato dell' esercizio precedente, cioe' la stampa di due lettere A separate da uno spazio bianco.

#### NOTA

Fino a questo momento sono state date le descrizioni complete dei registri, ma d' ora in poi, per brevit ,

li indicheremo con la sola lettera maiuscola e non piu' come REGISTRO INDICE X o Y, ma quindi come X o Y. Per esempio l' ultima istruzione vista TAX sara' descritta come:

TAX Trasferisci A in X.

Vediamo ora le altre istruzioni di trasferimento:

TAY Trasferisci A in Y

TXA Trasferisci X in A

TYA Trasferisci Y in A

#### Esercizio 4

Scrivere un programma che carichi una Z entro l' Accumulatore ed una A entro il registro X. Poi, senza utilizzare comandi in modo immediato, visualizzare la Z nella prima locazione di memoria di schermo e la A nell' ultima.

#### Esercizio 5

Scrivere un programma che carichi il simbolo dei quadri (DIAMOND) nell' Accumulatore, un asterisco in X ed una E in Y.

Senza usare i comandi in modo immediato, portare la E nell' Accumulatore, il simbolo dei quadri in X e l' asterisco in Y.

Visualizzare in lato a sinistra dello schermo il simbolo dei quadri, in basso a destra l' asterisco e nei rimanenti angoli liberi due E.

# LEZIONE TERZA

## I SALTI ED IL PROGRAM COUNTER

Come del resto avviene per il Basic, nessun programma procede attraverso una serie di passi ininterrotti, senza salti a subroutines, a Kernal routines o altro. Questo capitolo esamina questi comandi e il loro uso. Vedremo poi i flags che consentono di controllare i salti e le diramazioni.

### SALTI INCONDIZIONATI

Sono comandi che dicono al programma di saltare ad un certo indirizzo, semplicemente, cioè senza condizioni. Sul 6510 esistono solo due istruzioni di questo tipo. La prima che vediamo è:

**JMP** **JuMP** to the specified address

Cioè salta ad un dato indirizzo

Per esempio, **JMP 834** ordina di saltare alla locazione di memoria 834.

Questo potrebbe, e spesso è così, essere un modo di inserire pezzi di programma dimenticati o parti di programma in aggiunta.

Vediamo ora il comportamento in un programma in cui sia

stato effettivamente inserito l'istruzione detta.

Ora questo puo' essere scritto cosi':

#### PROGRAMMA 5

```
10*=828
20 LDA #1
30 JMP 834
40 RTS
50 STA 1024
60 STA 55296
70 JMP 833
```

Quando i salti sono usati in questo modo e' necessario dire al programma dove esattamente deve saltare, dando per questo un indirizzo come JMP 834.

Nel calcolo degli indirizzi di salto e' necessario tenere conto dell' occupazione di memoria derivata dall' uso dell' istruzione stessa ed eventualmente di quella degli operandi.

Vediamo, guardando per questo al programma precedente, la successione di memoria occupata dalle istruzioni in modo da comprendere bene perche' per superare il RTS e' necessario saltare a 834:

```
828 LDA #
829 1
830 JMP
831 - 832 (indirizzo 834)
833 RTS
834 STA
835 - 836 (indirizzo 1024)
```

```

837 STA
838 - 839 (indirizzo 55296)
840 JMP
841 - 842 (indirizzo 833)

```

Le parti seguenti i comandi, e che sono conosciute come operandi, determinano una qualche complicazione nel calcolo degli indirizzi.

Esiste una strada semplice o almeno relativamente semplice, per il calcolo degli indirizzi ed e' quella di adoperare le tavole in appendice dove sono riportati i codici delle istruzioni ed i bytes necessari.

JSR Jump to SubRoutines

Questo e' un' altro comando che consente un salto ad una Subroutine.

Utilizzandolo insieme a RTS consente una funzione simile a GOSUB.... RETURN del Basic.

Vediamo un esempio comparativo:

BASIC

ASSEMBLER

-----

-----

10 GOSUB 200

830 JSR 834

...

...

...

...

...

...

200 REM SUB ROUTINE

834 STA 1024

...

...

...

...

300 RETURN

840 RTS

Proviamo a modificare il programma 2.1 usando l'istruzione appena vista in luogo di JMP

### PROGRAMMA 6

```
10*=828
20 LDA #1
30 JSR 834
40 RTS
50 STA 1024
60 STA 55296
70 RTS
```

Il vantaggio di JSR su JMP e' dimostrato da questo programma dove con JSR non e' necessario calcolare l'indirizzo di salto per il ritorno al programma principale.

### PROGRAM COUNTER (PC)

Questo e' un registro a 16 bit che contiene gli indirizzi del prossimo comando da eseguire.

In realta' si tratta di due memorie di 8 bit ciascuna inserite entro il 6510.

Non appena il PC avanza, ogni Byte di memoria si incrementa di uno in modo tale da puntare alla successiva locazione di memoria che conterra' quindi i dati richiesti.

L'avanzamento del PC e' sequenziale per cui ad ogni incremento unitario, corrisponde un puntamento sulla locazione di memoria immediatamente successiva.

Prendiamo per esempio le prime tre linee del precedente programma:

```
10*=828
20 LDA #1
30 JSR 834
```

e vediamo un sommario dei contenuti del PC alla esecuzione delle varie istruzioni:

| PROGRAMMA  | CONTENUTO DEL PROGR. COUNTER |      |
|------------|------------------------------|------|
|            | PRIMA                        | DOPO |
| 10*=828    | ?                            | 828  |
| 20 LDA #1  | 828                          | 830  |
| 30 JSR 834 | 830                          | 834  |

Ricordiamo che questa area di memoria e' grande solo 16 bit per cui se e' necessario immagazzinare piu' di un indirizzo dovremo far ricorso ad un' area esterna di memorizzazione, lo STACK, che vedremo in seguito.

### Esercizio 7

Scrivere un programma che metta un 3 nell' Accumulatore. Il programma deve incominciare ad 828, saltare quindi (cioe' eseguire un JUMP) alla routine di indirizzo 900 che aggiunga un 3 al 3 gia' presente in A. Ritornare quindi alla routine originale e stampare, in alto a sinistra dello schermo, il contenuto dell' Accumulatore, cioe' il risultato della somma.

## SALTI CONDIZIONATI

Fino a questo momento abbiamo sempre visto dei salti incondizionati, ma un qualsiasi programma che necessiti di un minimo di controllo avra' la necessita' di SALTI CONDIZIONATI.

Per fare un' analogia con il Basic possiamo prendere il comando di condizionamento IF....THEN:

```
10 IF X=Y THEN 500
```

In questa linea i valori X e Y, che sono stati immagazzinati in memoria sono confrontati fra loro e se si verifica la condizione di eguaglianza, almeno in questo caso, si salta alla linea specificata dopo il THEN.

Il 6510 puo' eseguire questo tipo di operazione in una molteplicita' di modi.

Uno di questi e' attraverso l' uso di un particolare registro chiamato REGISTRO DI STATO o STATUS REGISTER (SR) o anche conosciuto come PROCESSOR STATUS WORD.

Lo STATUS REGISTER e' un registro di 8 bits come l' Accumulatore, i registri X e Y ma viene usato in maniera differente .

Mentre gli altri registri sono usati per immagazzinare e manipolare Bytes , questo registro divide e quindi considera separatamente i sui singoli BITS come FLAGS o segnali.

Di norma il 6510 manipola uno solo di questi Flags per volta, sia fissandone il suo valore a 0 oppure a 1 sia controllando se il valore e' a 0 oppure a 1.

In altre parole su questi Flags ( che sono poi i singoli bits del SR) si puo' , di volta in volta scrivere o leggere il valore relativo.

Un esempio di uno di questi Flags e' il flag Z o flag

ZERO.

Quando viene eseguito un programma o una manipolazione di dati che produca un risultato di 0 (zero) in un determinato registro (A,X o Y) allora il flag Z viene messo a 1. Se invece il risultato e' diverso da 0 allora lo Z flag viene messo a 0.

Altre istruzioni possono settare questo flag, una di queste e':

DEX DEcrement the contents of register X

Cioe' decrementa il contenuto del registro X.

Questo pezzo di programma ne dimostra l' uso.

PROGRAMMA 7 (Parte)

10 \*=828

20 LDX 100

30 DEX

Viene cioe' caricato X con 100 e poi decrementato, naturalmente di 1 fino a 99.

Quando il contenuto di X sia 0 (zero) allora il Flag ZERO viene messo a 1.

Se vogliamo usare questa capacita' del Flag per eseguire dei controlli sul programma, dovremo usare un' istruzione che controlli il contenuto del Flag stesso e che quindi consenta di effettuare salti o deviazioni in dipendenza del fatto che il Flag sia a 0 o a 1. Vediamone un' istruzione:

BEQ Branch if result was EQual to zero

Cioe' esegui il salto se il valore , per esempio del Flag Z, e' eguale a 0

E' necessario fare un po' di attenzione a questo punto per non confondersi.

Rileggendo quanto abbiamo detto in precedenza infatti, se il risultato dell' operazione e' ZERO allora il flag Z viene messo a 1, per cui il BEQ, cioe' la sua condizione, si verifica quando il flag Z e' =1.

L' operando, con questa istruzione e' solo di 1 Byte, per cui si possono manipolare numeri da 0 a 255.

Riprendiamo ora, sebbene in modo preliminare il concetto di LABEL.

Come abbiamo gia' accennato all' inizio nella presentazione dei comandi di questo Assembler per LABEL si intende un particolare indirizzo di memoria al quale abbiamo assegnato un nome.

Con l' istruzione BEQ PIPPO sara' effettuato un salto alla locazione di riferimento e cio' senza bisogno di contare il punto preciso a cui saltare.

#### PROGRAMMA 7

```

10 *=828
20 LDX #100
30 ROUT DEX
40 BEQ FINE
50 JMP ROUT
60 FINE STX 1024
70 STX 55296
80 RTS

```

Quando questo programma viene eseguito, verra' visualizzata una A commerciale nera in posizione 1024.

Come molte istruzioni relative al registro X anche DEX ha la sua istruzione corrispondente per il registro Y:

DEY DEcrement the contents of register Y.

Cioe' decrementa il contenuto del registro Y.

### Esercizio 8

Scrivere un programma simile al precedente ma che utilizzi pero' il registro Y.

Un' altra istruzione che controlla lo stato del Flag Z e' la seguente:

BNE Branch if Not Equal

Cioe' salta se non eguale

Questa istruzione e' esattamente il rovescio della BEQ ed esegue il salto se il Flag Z e' a 0, cioe' non e' settato.

Il seguente programma e' una modifica del programma precedente.

Notate come questo programma sia un po' piu' corto del precedente che faceva uso dell' istruzione BEQ.

### PROGRAMMA 8

```
10 *=828
20 LDX #100
30 ROUT DEX
40 BNE ROUT
50 STX 1024
```

```
60 STX 55296
70 RTS
```

Il risultato sara' identico a quello ottenuto in precedenza con il programma 7.

I registri indice X e Y sono stati decrementati o INDICIZZATI VERSO IL BASSO, cioe' verso un valore inferiore, con le istruzioni DEX e DEY e naturalmente e' possibile eseguire l' operazione opposta o indicizzarli verso l' alto, cioe' incrementarli, utilizzando le seguenti istruzioni:

INX INcrement the contents of X by 1

Cioe' incrementa il contenuto del registro X di 1

INY INcrement the contents of Y by 1

Cioe' incrementa il contenuto del registro Y di 1

## LEZIONE QUARTA

### ISTRUZIONI DI CONFRONTO

Utilizzando gli incrementi un vero controllo per il valore 0 non e' naturalmente possibile per cui i registri devono essere confrontati con un valore preventivamente immagazzinato da qualche parte. Il 6510 ha 3 istruzioni per eseguire questo controllo.

CPX. ComPare the contents of the specified memory address with the X register.

Cioe' confronta il contenuto di un dato indirizzo di memoria con il registro X.

Questo viene fatto sottraendo il contenuto di memoria da X cosa che puo' dare un valore positivo, negativo o zero.

Percio' l'istruzione CPX 900 si comportera' nel modo seguente:

- 1) Legge il contenuto della locazione di memoria 900
- 2) Sottrae questo contenuto da quello del registro X
- 3) Mette a 1 ( o setta) il flag Z se il risultato e' =0

### NOTA

Ne' il contenuto della locazione di memoria 900 ne' il

contenuto di X vengono pero' cambiati durante questa fase.

Per il momento noi siamo interessati alla condizione zero ( infatti anche altri flags sono interessati da operazioni simili).

Per utilizzare questa istruzione possiamo mettere il registro X a zero ed immagazzinare un valore di confronto da qualche parte della memoria.

Vediamone un' applicazione in programma commentata.

#### PROGRAMMA 9

```

10  *=828
20  LDA #90
30  STA 890
40  LDX #0
50  ROUT INX
60  CPX 890
70  BEQ FINE
80  JMP  ROUT
90  FINE STX 1024
100 LDA #1
110 STA 55296
120 RTS
    
```

#### COMMENTO

```

20  Carica 90 in Accumulatore
30  Carica il contenuto di A a 890
40  Carica 0 nel registro X
50  Incrementa il registro X
60  Confronta il valore in X con quello in 890
70  Vai avanti di 3 Bytes se dal confronto risulta CPX=0
80  Vai a 835
90  Immagazzina il contenuto di X in 1024
    
```

100 Carica 1 in Accumulatore  
110 Immetti il valore dell' Acc. (1) in RAM per dare il colore bianco.

Naturalmente l' istruzione CPX ha la sua corrispondente in Y:

CPY Compare the contents of specified location with those in the Y register.

Cioe' confronta il risultato di una specifica locazione di memoria con il valore contenuto nel registro Y.

Il cui risultato e commento e' quindi esattamente eguale a quello visto in precedenza per X.

#### Esercizio 9

Riscrivere il programma precedente usando il registro Y ed alla fine del ciclo far stampare a 1034 un cuore porpora.

Ricordarsi che il colore porpora viene dato dal valore 4 in RAM anziche' da 1 come per il bianco.

La terza istruzione di confronto e':

CMP CoMPare the contents of the specified with the Accumulator.

Cioe' confronta il contenuto di una particolare locazione di memoria con l' Accumulatore.

Questa istruzione e' particolarmente utile perche' il risultato di tutte le operazioni aritmetiche e'

## CORSO DI ASSEMBLER CBM64

depositato nell' Accumulatore e quindi CMP consente un confronto diretto fra un dato valore e la "risposta". Un esempio di cio' viene fornito nel programma seguente:

### PROGRAMMA 10

```
10  *=828
20  LDX #0
30  LDA #83
40  ROUT INX
50  STX 900
60  CMP 900
70  BNE ROUT
80  STX 1024
90  LDA #1
100 STA 55296
110 RTS
```

### COMMENTO

```
20  Carica 0 in X
30  Carica 83 (il cuore) in A
40  Incrementa X
50  Immagazzina X in 900
60  Confronta A con 900
70  Salta se diverso
80  Immagazzina X in 1024
90  Carica 1 in A
100 Immetti 1 in 55296 per il colore bianco
```

**I FLAGS DEL 6510**

Fino a questo momento abbiamo lavorato solo su 1 dei 7 Flags disponibili sul 6510.

Vediamo quali sono gli altri

|        |   |   |   |   |   |   |   |   |
|--------|---|---|---|---|---|---|---|---|
| BIT N. | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| FLAG   | N | V | - | B | D | I | Z | C |

Diamo ora un sommario di questi Flags che vedremo uno per uno, come abbiamo fatto per Z, nel corso del manuale.

**N FLAG NEGATIVE.**

Viene settato o messo a 1 quando il risultato di un' operazione aritmetica e' negativo

**V OVERFLOW FLAG.**

Viene settato quando i risultati di un' operazione aritmetica vanno in overflow dal bit 6 al 7.

**B BREAK FLAG**

Viene messo a 1 quando avviene un' interruzione di programma messa in funzione da un' istruzione BRK.

**D DECIMAL FLAG**

A 1 quando si opera in modo decimale

**I INTERRUPT FLAG**

Viene messo a 1 quando opera una sequenza di Interrupt.

## Z ZERO FLAG

E' stato abbondantemente spiegato

## C CARRY FLAG

Indica la presenza di un Carry, cioe' di un riporto durante un' operazione aritmetica.

Messo a 1 anche durante le operazioni di SHIFT o ROTATE per indicare la possibile perdita di un bit.

## IL FLAG N

Questo Flag, cioe' il NEGATIVE FLAG, viene messo a 1 quando la risposta a un' operazione e' un risultato negativo.

Puo' essere controllato da due istruzioni:

BMI Branch on Minus

Un' istruzione come BMI 27 eseguirà il controllo sul flag N e se questi e' a 1 allora salterà in avanti di 27 Bytes.

Un esempio d' uso di BMI e' dato nel programma seguente, in cui il contenuto di Y e' incrementato fino a quando un comando CPY da un meno e si passa all' istruzione BMI:

## PROGRAMMA 11

```

10  *=828
20  LDA #90
30  STA 900
40  LDY #0
50  ROUT INY
60  CPY 900
70  BMI ROUT
80  STY 1024
90  LDA #1
100 STA 55296
110 RTS

```

## COMMENTO

```

20  Carica 90 in A
30  Immagazzina il contenuto di A in 900
40  Carica 0 in Y
50  Incrementa Y
60  Confronta il contenuto di 900 con il contenuto di Y
70  Controlla il flag N
80  Immetti il contenuto di Y in 1024
90  Carica 1 in A
100 Metti il colore in RAM

```

In aggiunta al comando BMI il flag N può essere controllato da:

BPL Branch on PLus

Un'istruzione come BPL 12 controllerà il contenuto del Flag N e se questi non è a 1 eseguirà un salto di 12

Bytes.

Vediamone un' applicazione nel seguente programma:

# PROGRAMMA 12

```

10  *=828
20  LDA #91
30  STA 900
40  LDY #100
50  ROUT DEY
60  CPY 900
70  BPL ROUT
80  STY 1024
90  LDA #7
100 STA 55296
110 RTS

```

Il risultato dell' esecuzione di questo programma sara' un quadri (90) giallo (7) in 1024.

## Esercizio 10

Scrivere un programma usando BPL per saltare quando il registro X arriva a 0 essendo stato decrementato a partire da 90.

A questo punto visualizzare l' attuale valore di X.

# LEZIONE QUINTA

## ALTRE ISTRUZIONI

Uno dei vantaggi della programmazione in codice macchina e' la velocita' di esecuzione e questo naturalmente facilita la visualizzazione dei risultati. Le animazioni, i giochi ad esempio, possono essere migliorate, come velocita' esecutiva, usando un comando come:

STA (LOC),X    STore the contents of Accumulator in the specified address index with the X register

Cioe' immagazzina il contenuto dell' Accumulatore in una locazione di memoria indicizzato dal contenuto del registro X.

Questo vuol dire che se X contiene 100 e A 90, l'istruzione :

STA 1024,X

immetterà il simbolo dei quadri in (1024+100).

Quando si usa questo comando incrementando l' indice si può indicizzare una qualsiasi locazione, in questo caso quella di schermo.

Vediamone una dimostrazione con il programma 13.

## PROGRAMMA 13

```

10 *=828
20 LDX #100
30 ROUT LDA #90
40 STA 1023,X
50 LDA #1
60 STA 55295,X
70 DEX
80 BNE ROUT
90 RTS

```

## COMMENTO

```

20 Carica 100 in X
30 Carica 90 in A
40 Visualizza a (1023+X)
50 Carica 1 in A
60 Si assicura che il colore sia il BIANCO
70 Decrementa il valore di X
80 Branch se diverso
90 Ritorno da subroutine

```

Quando questo programma gira, mette il quadri nelle prime 100 locazioni di schermo.

Naturalmente il comando visto per X ha il suo corrispondente nell' uso del registro Y.

STA (LOC),Y [ STore the contents of Accumulator] in the specified address indexed with the Y register.

Cioe' immagazzina il contenuto dell' Accumulatore in una locazione di memoria indicizzato dal contenuto del

registro Y.

### Esercizio 11

Modificare il programma 13 usando il registro Y invece di X. Usare solo comandi diretti di POKE.

### Esercizio 12

Stampare un asterisco nelle prime 100 locazioni di schermo usando un comando di incremento INX.

### NOTA

Si consiglia di cercare di risolvere questo esercizio prima di proseguire.

Il programma qui sotto (14) consente una funzione simile a quella vista con il programma 13 ma usa il INX, cioè l' incremento, invece di DEX.

Questo programma di per se non offre particolari guadagni rispetto al precedente, ma in alcune situazioni puo' essere piu' vantaggioso.

### PROGRAMMA 14

```
10 *=828
20 LDX #216
30 ROUT LDA #42
40 STA 808,X
50 LDA #1
```

## CORSO DI ASSEMBLER CBM64

```
60 STA 55080,X
70 INX
80 BNE ROUT
90 RTS
```

Vediamo ora un programma che puo' essere usato anche come routine da aggiungere ad altri programmi, che serve a muovere un carattere sullo schermo. Questo risultato puo' essere ottenuto in programmi Basic con una serie di istruzioni POKE.

### PROGRAMMA 15

```
10 *=828
20 LDX #0
30 LDY #32
40 STY 900
50 LDA #90
60 STA 901
70 INCR STA 1024,X
80 LDA #1
90 STA 55296,X
100 TYA
110 STA 1023,X
120 LDA 901
130 INX
140 BNE INCR
150 RTS
```

Quando gira, il programma esposto fa muovere il simbolo dei quadri , in bianco, lungo lo schermo fino a 1279.

La tavola sottoriportata e' utile come CROSS-REFERENCE

per vedere passo passo il programma.

| Comando     | Acc | Re.X | Re.Y | Mem. | Co. | Z |
|-------------|-----|------|------|------|-----|---|
| LDX #0      | ?   | 0    | ?    | ?    | ?   | 0 |
| LDY #32     | ?   | 0    | 32   | ?    | ?   | 0 |
| STY 900     | ?   | 0    | 32   | ?    | ?   | 0 |
| LDA #90     | 90  | 0    | 32   | ?    | ?   | 0 |
| STA 901     | 90  | 0    | 32   | ?    | ?   | 0 |
| STA 1024,X  | 90  | 0    | 32   | 1024 | 90  | 0 |
| LDA #1      | 1   | 0    | 32   | 1024 | 90  | 0 |
| STA 55296,X | 1   | 0    | 32   | 1024 | 90  | 0 |
| TYA         | 32  | 0    | 32   | 1024 | 90  | 0 |
| STA 1023,X  | 32  | 0    | 32   | 1023 | 90  | 0 |
| LDA 901     | 90  | 0    | 32   | 1023 | 32  | 0 |
| INX         | 90  | 1    | 32   | 1023 | 32  | 0 |
| BNE 238     | 90  | 1    | 32   | 1023 | 32  | 0 |
| RTS         | 90  | 0    | 32   | 1278 | 32  | 1 |

Come abbiamo detto in precedenza, il programma 15, pur essendo uno dei tanti sistemi con il quale si puo' scrivere un programma, forse non e' il migliore, pur funzionando, cioe' risolvendo lo scopo per il quale e' stato scritto.

Successivamente in questo capitolo ne vedremo una versione migliore.

Vediamo ora invece un problema legato anche a questo tipo di programma: La temporizzazione.

## LA TEMPORIZZAZIONE DEI PROGRAMMI

Il programma 15 mostra con notevole efficacia uno dei problemi della programmazione in codice macchina, la velocità'.

Mentre, di solito in Basic non è quasi mai necessario diminuire la velocità' che è già' lenta di suo, altrettanto non può' dirsi per quanto riguarda il Codice Macchina.

Il microprocessore 6510 ricava la sua VELOCITA' OPERATIVA da un clock interno ( un oscillatore al cristallo di quarzo molto preciso) che nel caso del CBM64 gira a 2 MHz (due MegaHertz) o due milioni di cicli al secondo.

Così' ogni ciclo richiede mezzo-milionesimo di secondo e la velocità' operativa delle varie istruzioni sarà' riferita al numero di cicli necessari per la loro esecuzione.

Alcune di queste operazioni prendono un posto entro il microprocessore e sono eseguite in maniera molto più' veloce di altre che invece devono andare a prendere dati dalla memoria.

Per esempio l'istruzione TAX prende 2 cicli, mentre per eseguire STA (L),X ne sono necessari 6.

Naturalmente la conoscenza del tempo richiesto per l'esecuzione del ciclo di istruzione è' importante per determinare la velocità' operativa del programma e consente di usare correttamente il clock da 2MHz per i cicli e per i ritardi programmati.

Ritornando a vedere l'esecuzione del programma 15 può' essere calcolato il tempo che il simbolo (QUADRI) rimane sullo schermo.

La tavola sotto riporta le istruzioni del programma, i cicli esecutivi e la sommatoria dei tempi, relativamente alla visualizzazione del carattere e quindi alla

effettiva esecuzione di quella parte di programma.

| COMANDO     | TEMPO/CICLO | SOMMAT.TEMPI |
|-------------|-------------|--------------|
| STA 1024,X  | -           | 0            |
| LDA #1      | 2           | 2            |
| STA 55296,X | 5           | 7            |
| TYA         | 2           | 9            |
| STA 1023,X  | 5           | 14           |
| LDA 901     | 4           | 18           |
| INX         | 2           | 20           |
| BNE 243     | 2           | 22           |
| STA 1024,X  | 5           | 27           |
| LDA #1      | 2           | 29           |
| STA 55296,X | 5           | 34           |
| TYA         | 2           | 36           |
| STA 1023,X  | 5           | 41           |

Così vediamo che dal momento dell'apparizione del carattere al momento in cui lo stesso è cancellato (o sovrascritto da un BLANK che è la stessa cosa), sono necessari o passano 41 cicli per un totale di 20.5 microsecondi.

I 256 caratteri saranno perciò scritti in 5248 micro-secondi o 5.2 circa milli-secondi. Un tempo chiaramente troppo breve perché l'occhio umano possa seguirlo.

Per avere quindi qualcosa di visibile è necessario programmare un ritardo (DELAY).

Il programma seguente mostra un semplice ciclo di ritardo.

## PROGRAMMA 16

|    |           |         |
|----|-----------|---------|
| 10 | *=828     |         |
| 20 | LDX #250  | 2 cicli |
| 30 | DECRX DEX | 2 "     |
| 40 | BNE DECRX | 3 "     |
| 60 | RTS       |         |

Cio' da un ritardo di 5 cicli per giro ( ignorando i 2 cicli per l'istruzione LDX #) o  $250 \times 5 = 1250$  cicli per esecuzione.

Dopo aver provato che giri correttamente si puo aumentare il ritardo , al momento di 625 micro-secondi, inserendo in questo programma di ritardo altre istruzioni o congiungendolo con un altro programma di ritardo come mostrato di seguito.

## PROGRAMMA 17

|    |               |         |
|----|---------------|---------|
| 10 | *=828         |         |
| 20 | LDY #200      | 2 cicli |
| 30 | RIT1 LDX #250 | 2 "     |
| 40 | RIT2 DEX      | 2 "     |
| 50 | BNE RIT2      | 3 "     |
| 60 | DEY           | 2 "     |
| 70 | BNE RIT1      | 2 "     |
| 80 | RTS           |         |

Quando gira completamente la subroutine DEX dovrebbe dare un ritardo aggiuntivo di  $200 \times 625$  micro-secondi o 1/8 di secondo.

Qualora si desideri usare il computer come

temporizzatore di precisione o comunque quando si abbia necessita' di misurare il tempo con assoluta precisione e' chiaro che non si puo' mettere un ritardo qua e la ne ignorare i 2 microsecondi come abbiamo fatto per semplificare il problema per l'istruzione LDX #.

Sara' invece vitale assicurarsi l' assoluta certezza del calcolo dei tempi.

In particolare e' necessario fare attenzione alle istruzioni di BRANCH.

L' istruzione BNE nel programma 3.5 normalmente necessita di tre cicli; per esempio quando il BRANCH ha successo o viene eseguita.

Tuttavia quando non viene eseguita e quindi il programma passa oltre sono necessari solo 2 cicli.

In altre condizioni se il BRANCH rimanda il programma ad altra pagina di memoria sono necessari due cicli addizionali.

Un altro problema poi tipico del CBM 64 e' che l' integrato VIC-II che controlla la visualizzazione di schermo puo' interferire con la temporizzazione.

In particolare quando si manipola la grafica e specialmente gli SPRITES il VIC-II ha veramente molto lavoro da compiere.

Spesso infatti deve prendere il controllo completo delle operazioni condotte ed allora al 6510 non resta che attendere.

Cio' porta alla conclusione che se realmente si desidera un' accurata temporizzazione e' necessario arrestare l'attivita' di visualizzazione del VIC-II, eseguire il lavoro o la parte di programma che necessita di tempi precisi e fare ripartire dopo il lavoro di visualizzazione.

Queste operazioni non sono pero' particolarmente difficili in Assembler.

Tuttavia in questo momento siamo interessati in modo particolare a ritardi di animazioni sullo schermo e

continuiamo a vedere sia come operano sia come si inseriscono in programma.

Il programma 18 usa il precedente 15 come base di lavoro ed inserisce il ciclo di ritardo illustrato in 16 immettendo un tempo di 0.6 millisecondi fra la visualizzazione e la cancellazione del carattere.

## PROGRAMMA 18

```
10  *=828
20  LDY #0
30  LDA #90
40  STA 900
50  LDA #1
60  STA 901
70  LDA #32
80  STA 902
90  INCREM LDX #250
100 LDA 900
110 STA 1024,Y
120 LDA 901
130 STA 55296,Y
140 RIT DEX
150 BNE RIT
160 LDA 902
170 STA 1024,Y
180 INY
190 BNE INCREM
200 RTS
```

Provate ad inserire questo programma ed a farlo girare.

Non riuscite a vedere niente!!!

La verita' e' che 0.6 millisecondi non bastano.

Lo schermo televisivo ha un REFRESH di 1/50 di secondo (

sistema europeo PAL ) o di 1/60 di secondo ( sistema americano NTSC ) cosi' che per la scansione di schermo sono necessari 16/20 millisecondi.

Se il nostro carattere bianco e' sullo schermo solo per un terzo di questo tempo, vorra' dire che vedremo SOLO un terzo delle immagini che si desiderava visualizzare.

### NOTA

In effetti a causa del cosi' detto INTERLACE la situazione e' leggermente migliore, cioe' si vedono un po' piu' di immagini di quelle teoricamente visibili. Tuttavia cio' non risolve il problema.

In ogni caso quindi il ritardo prodotto non e' sufficiente e va aumentato. Come?

Il registro X puo' manipolare un massimo di 255 per cui l' unica soluzione e' di far ricorso, nello stesso modo che si farebbe con un programma in Basic, ad una Subroutine di temporizzazione o, nel caso non sia sufficiente, a piu' routines.

### Programma 19

```
10  *=828
20  LDY #0
30  LDA #90
40  STA 900
50  LDA #1
60  STA 901
70  LDA #32
80  STA 902
100 INCLOC LDA 900
110 STA 1024,Y
120 LDA 901
130 STA 55296,Y
```

## CORSO DI ASSEMBLER CBM64

```
135 STY 903
136 LDY #15
137 LOOPA LDX #250
140 LOOPB DEX
150 BNE LOOPB
155 DEY
156 BNE LOOPA
157 LDY 903
160 LDA 902
170 STA 1024,Y
180 INY
190 BNE INCLOC
200 RTS
```

Inserire questo programma e farlo girare.

Questo dimostra il motivo per cui i programmi giochi scritti in codice macchina funzionano in modo tanto piu' veloce e quindi migliore di quelli scritti in Basic.

### NOTA

Per approfondimenti sia sulla temporizzazione che sul funzionamento dei circuiti Hardware si consiglia il manuale LE PERIFERICHE COMMODORE ed EVM.

# LEZIONE SESTA

## MODI DI INDIRIZZAMENTO

Quando abbiamo mosso un carattere lungo lo schermo sono state usate le istruzioni STA, X e STA, Y che sono in grado di indicizzare un numero importante di registri. In parole semplici si usa l'istruzione STA su se stessa e cio' e' chiaramente un modo di dire di STA,X STA,Y ecc.

La differenza fra i due modi di istruzioni sta nel loro indirizzamento.

In effetti il comando STA ha numerosi modi che dipendono dall'indirizzamento usato.

Si puo' quindi dire che l'indirizzamento e' una modifica del comando fatta per cambiare la sua funzione in modo particolare.

L'indirizzamento fa si che il 6510 punti ( o sia puntato o indirizzi ) ad una locazione di memoria sia direttamente che indirettamente.

La strada seguita dipende dal modo particolare di indirizzamento usato.

Gli indirizzamenti sono uniformi attraverso tutti i 64 K di memoria disponibile tranne che per i primi 256 Bytes di memoria ( dalla locazione 0 alla 255 ).

Per indirizzare queste locazioni ( o prima pagina di memoria ) e' necessario solo 1 Byte mentre tutte le altre pagine necessitano di 2 Bytes.

### NOTA

Ricordiamo che l'intera mappa di memoria del CBM64 puo' essere divisa in pagine ognuna delle quali di 256 Bytes e che la prima pagina e' chiamata appunto PAGINA ZERO che ha uno speciale modo di indirizzamento che vedremo nel seguito di questo capitolo.

Ricordiamo inoltre che quando in risposta ad un comando si salta da una pagina all'altra a livello di temporizzazione verra' usato un ciclo addizionale.

### INDIRIZZAMENTO IMMEDIATO

Questo modo di indirizzamento consente che un numero sia caricato immediatamente entro un registro o per essere usato direttamente come termine di un confronto.

Questo modo di indirizzamento e' identificato chiaramente dal segno POUND (#) che precede il valore che deve essere caricato.

LDA #10 e' un modo immediato di indirizzare l'Accumulatore e consente che il valore immediatamente seguente sia caricato nell'accumulatore stesso.

La corrispondente istruzione Basic potrebbe essere A=10.

Questo modo e' utilizzato per caricare un registro con una costante e puo' lavorare anche con i registri X e Y. Quando si impiega questo modo di indirizzamento il valore che deve essere caricato e' parte del programma.

In altre parole l'istruzione ed il valore sono immessi uno dietro l'altro in due adiacenti locazioni di memoria.

Fino ad ora sono stati visti numerosi esempi del modo di

## CORSO DI ASSEMBLER CBM64

INDIRIZZAMENTO IMMEDIATO come per esempio nel programma 18.

In questo programma l' Accumulatore era caricato direttamente usando LDA #32, mentre in altri programmi sia il registro X che il registro Y sono stati caricati con lo stesso sistema.

Anche altre istruzioni possono essere usate in modo immediato come vediamo nel programma di seguito.

Questo programma mostra inoltre l' uso di una nuova istruzione:

CPY # ComPare Y with value specified in Immediate Mode.

Cioe' confronta Y con il valore specificato in Modo Immediato.

### PROGRAMMA 20

```
10  *=828
20  LDY #0
30  INCLOC TYA
40  INY
50  STA 1023,Y
60  LDA #1
70  STA 55295,Y
80  CPY #100
90  BNE INCLOC
100 RTS
```

### COMMENTO

```
20  Carica Y con 0
30  Trasferisci Y in A inizio ciclo
40  Incrementa Y
50  Immagazzina il contenuto di A in 1023+Y
```

## CORSO DI ASSEMBLER CBM64

```
60 Carica A con 1
70 Immagazzina A in 55295 + Y
80 Confronta Y con 100
90 Branch se il flag Z non e' stato settato (fine
ciclo)
100 RTS
```

Quando gira questo programma stampa i primi 100 caratteri del set di caratteri in memoria sulle prime 100 locazioni di schermo

### INDIRIZZAMENTO IMPLICITO

Questo modo, chiamato qualche volta anche indirizzamento inerente, e' probabilmente il piu' facile da usare in quanto e' il 6510 ad eseguire tutto il lavoro.

Con questo modo possono essere utilizzate numerose istruzioni come TYA, TXA, RTS in quanto il 6510 stesso calcola gli indirizzi.

Fondamentalmente le istruzioni possono dividersi in due gruppi separati.

Nel primo gruppo possono essere messe le istruzioni che sono eseguite interamente entro il 6510 come TYA che trasferisce Y in A e quindi tutto avviene all' interno del microprocessore.

Nel secondo gruppo possiamo mettere invece le istruzioni dove e' necessario un riferimento esterno come per esempio RTS.

Le istruzioni del primo gruppo sono:

DEX DEY INX INY TAX TXA TYA CLC CLD CLI CLV NOP SEC SED  
SEI.

Quelle del secondo gruppo:  
RTS BRK PHA PHP PLA PLP RTI

### INDIRIZZAMENTO ASSOLUTO

Le istruzioni usate in questo modo sono facili da comprendere in quanto l'operando dell'istruzione ( il numero che viene accanto all'istruzione stessa ) e' un numero di 2 Bytes che definisce appunto l'indirizzo in modo assoluto.

In questo modo, per esempio, nel programma del capitolo precedente l'istruzione STA 901 comunica al registro A ESATTAMENTE dove immagazzinare il suo contenuto.

Le istruzioni che utilizzano questa forma di indirizzamento sono elencate di seguito ed in parte sono gia' state viste mentre altre le vedremo in seguito:

ADC CMP CPX CPY JMP JSR LDA LDX LDY STA STX STY AND EOR  
ORA SBC

### INDIRIZZAMENTO IN PAGINA ZERO

Questa forma di indirizzamento e' in realta' una sotto-forma dell'indirizzamento assoluto solo che l'operando e' ristretto ad un Byte cioe' massimo 256 caratteri.

Il maggior vantaggio di questo tipo di indirizzamento e' la velocita' di esecuzione perche' le istruzioni sono eseguite in soli tre cicli invece che in quattro come

## CORSO DI ASSEMBLER CBM64

nell' indirizzamento assoluto normale.

A causa della maggior velocita' di esecuzione la pagina zero e' adoperata quasi per intero dal Sistema Operativo e dall' interprete BASIC per cui non e' realmente disponibile per l' Assembler.

Al momento si possono utilizzare le locazioni di pagina zero da 251 a 254.

Quando ne saprete di piu' sull' interprete Basic potrete utilizzare altre locazioni di questa pagina ed addirittura spostare la pagina zero con il suo contenuto da altre parti della memoria.

Considerazioni piu' approfondite esulano pero' dagli scopi di questo manuale.

Malgrado sia pericoloso utilizzare le locazioni di questa pagina si possono pero' leggere ed utilizzare le informazioni qui contenute.

Tre locazioni utili di questa pagina possono essere la 160, 161 e 162 che contengono il valore del clock o JIFFIES CLOCK o OROLOGIO ( espresso in ore, minuti e secondi ) che si incrementa di un 1/60 di secondo.

Il programma 21 e' un semplice programma che carica uno di questi valori in A e lo stampa sullo schermo.

### PROGRAMMA 21

```
10 *=828
20 LDA 160
30 STA 1024
40 LDA #1
50 STA 55296
60 RTS
```

## INDIRIZZAMENTO INDICIZZATO ASSOLUTO

In questo modo un indirizzo viene calcolato usando il contenuto di un registro aggiunto ad un dato indirizzo. E' stato usato di frequente per stampare caratteri sullo schermo con la forma `STA LOC,X` e `STA LOC,Y`. Nel programma 20 `STA LOC,Y` era stato usato in questo modo con il comando `STA 1024,Y`

Quando si usa questo sistema di indirizzamento bisogna fare attenzione perche' il modo di comportarsi del registro X rispetto al registro Y e' diverso. Entrambi i registri possono essere usati con istruzioni di indicizzamento assoluto, per esempio operando con due Bytes.

I codici mnemonici che devono essere usati in pagina ZERO devono avere l'indirizzo della pagina che sara' costituito da un solo Byte.

## INDIRIZZAMENTO RELATIVO

Molti programmi usati fino a questo momento hanno utilizzato indirizzi relativi, nei quali un salto e' stato definito relativamente all'attuale posizione del programma.

Per esempio l'operando che esprime la posizione desiderata.

Tutte le istruzioni di salto usate in questo modo utilizzano l'indirizzamento relativo.

Il gruppo e' composto dai seguenti comandi:

`BCC BCS BEQ BMI BNE BPL BVC BVS`

### INDIRIZZAMENTO INDIRETTO

Questo e' allo stesso tempo il piu' complesso ed il piu' versatile di tutti i modi di indirizzamento.

Questo modo prende il nome di INDIRETTO dal fatto che l'operando e' un puntatore e non un indirizzo.

Ed e' questo puntatore che dirige il 6510 attraverso le locazioni di memoria che contengono l'indirizzo.

Ancora una volta tuttavia i meccanismi di indicizzazione di X e Y differiscono fra loro in misura considerevole e danno luogo a diversi modi di indirizzamento.

Tutte le istruzioni che utilizzano questo metodo sono riconoscibili in assembler perche' contengono o un suffisso IX o IY ed hanno un operando di 1 Byte

A causa di cio' possono puntare solo a locazioni in pagina zero e percio' sono sottoposte alle stesse restrizioni gia' viste per gli altri comandi in pagina zero.

### USO DEL REGISTRO X

Con un indirizzamento indiretto che usi il registro X, l'operando e' indicizzato (aggiunto) con il contenuto dello stesso registro per produrre il puntatore.

Questa locazione e quella immediatamente successiva sono quindi esaminate ed i loro contenuti forniscono gli indirizzi per i dati richiesti con l'ordine seguente:

Byte meno significativo (LSB)

Byte piu' significativo (MSB)

Questa tecnica e' utile per esaminare un particolare elemento in una tavola, essendo fissato l' attuale posizione della tavola dal valore del registro X.

Data la scarsa disponibilita' dello spazio sulla pagina zero sul CBM64 il modo di indirizzamento e' di uso limitato, tuttavia, a scopo dimostrativo, faremo vedere un programma dove viene usata una istruzione di questo modo.

L' istruzione e':

LDA (LOC),X Load A Indirectly indexed with X

Cioe' carica l' Accumulatore con l' indirizzo indiretto indicizzato con il contenuto di X.

Nel nostro caso e' usata per trovare 4 bytes immagazzinati in pagina ZERO da 84 a 88.

#### PROGRAMMA 22

```
10 *=828
20 LDX #0
30 LOOP LDA (84,X)
40 STA 1024,X
50 LDA #1
55 STA 55296,X
60 INX
70 CPX #4
80 BNE LOOP
90 RTS
```

#### COMMENTO

20 Carica X immediato con 0

## CORSO DI ASSEMBLER CBM64

```
30 Carica A indiretto 84+X
40 Immagazzina A in 1024+X
50 Carica A con 1
55 Mette il colore nella locazione di schermo
60 Incrementa X
70 Esegui un confronto immediato di X con 4
80 Torna a LOOP
```

Quando questo programma gira, saranno visualizzati 4 caratteri nelle prime 4 locazioni di schermo.

Questi caratteri differiranno in base a cio' che stava facendo il Basic per ultimo.

Quando si usa questa routine in un programma i quattro numeri dovrebbero formare due indirizzi con il seguente ordine:

|             |             |     |
|-------------|-------------|-----|
| Carattere 1 | Indirizzo 1 | LSB |
| " 2         | " 1         | MSB |
| " 3         | " 2         | LSB |
| " 4         | " 2         | MSB |

Questo tipo di indirizzamento e' conosciuto come INDIRIZZAMENTO INDICIZZATO INDIRETTO o, molto piu' chiaramente INDIRIZZAMENTO PREINDICIZZATO INDIRETTO.

Infatti, come e' implicito nel nome stesso, questo indirizzamento e' preindicizzato poiche' il valore di X e' aggiunto prima che il 6510 salti all' indirizzo.

### USO DEL REGISTRO Y

Usando l' indirizzamento indiretto con il registro Y si opera in modo differente, poiche' l' istruzione operando punta direttamente ad una locazione di memoria in pagina

zero.

Questa contiene il LSB dell' indirizzo e la successiva locazione di memoria contiene il MSB.

Finalmente il contenuto indicizzato del registro e' aggiunto a questo indirizzo per formare l' indirizzo finale indicizzato.

Non deve sorprendere quindi se questa forma e' chiamata anche INDIRIZZO INDIRETTO POSTINDICIZZATO in quanto l' indicizzazione e' calcolata DOPO che l' indirizzo e' stato trovato.

L' interprete Basic ed il Sistema Operativo del CBM64 fanno un uso molto esteso di questa istruzione.

Quando avrete una maggiore confidenza con l' uso dell' Assembler potrete vedere come lavori il Basic e trarne notevole vantaggio nell' uso delle routines del Basic stesso ed in generale del Sistema Operativo del computer.

Vediamo un breve esempio

```
10 *=828
20 LDY #0
30 LDA (43),Y
40 STA 1024,Y
50 LDA #1
60 STA 55296,Y
70 RTS
```

Questo programma prelevera' il valore presente nella locazione indirizzata dai valori presenti in pagina zero agli indirizzi 43 e 44 (in questo caso l' inizio Basic cioe' 2048) e lo visualizza nella prima locazione di schermo.

Ricordiamo che nel volume IL SISTEMA OPERATIVO DEL CBM 64 viene riportato l' INTERO Ssistema Operativo del CBM 64 completamente disassemblato e commentato.

## INDIRIZZAMENTO INDIRETTO ASSOLUTO

Questo modo di indirizzamento e' usato con una sola istruzione:

JMP (LOC) JuMP Indirectly Addressed

Cioe' salta ad un indirizzo indiretto

E' questa un' istruzione assoluta nel quale l' operando e' un indirizzo di 2 bytes e puo' quindi indirizzare una qualsiasi locazione di memoria.

E' tuttavia indiretto in quanto a quella locazione ed a quella successiva trova l' indirizzo (prima LSB e poi MSB) per l' istruzione di salto.

Vediamo un esempio:

JMP (\$A000)

in questo caso eseguirà un salto all' indirizzo di memoria puntata dalle locazioni \$A000 e \$A001 che in questo caso e' \$E394.

# LEZIONE SETTIMA

## OPERAZIONI MATEMATICHE

Nelle prime pagine ed in particolare nel primo programma presentato avevamo fatto un esempio di somma e di visualizzazione del risultato.

La semplicità del programma e dei numeri da sommare veniva dal fatto che erano numeri decimali inferiori a 256 e che la risposta non richiedeva il riporto.

Quando è necessario operare su numeri di dimensioni maggiori allora il 6510 li manipola usando il suo riporto o CARRY o C FLAG.

Usando un Byte è possibile contare solo fino a 255, per cui se vogliamo contare oltre dobbiamo usare due Bytes. Questi 16 Bits consentono allora di contare da 0 a 65535.

È possibile manipolare naturalmente numeri di dimensioni molto più grandi di questi, tuttavia per il momento ci limiteremo a descrivere operazioni con solo due Bytes che vengono chiamate:

## OPERAZIONI IN DOPPIA PRECISIONE

Se due o più Bytes devono essere utilizzati per rappresentare uno stesso numero allora si deve creare un legame (LINK) fra il primo ed il secondo Byte tramite un meccanismo univoco.

## CORSO DI ASSEMBLER CBM64

Questa e' la funzione del CARRY.

Il suo funzionamento e' provato dall' istruzione seguente:

BCC Branch on Carry Clear

Questa istruzione controlla che il CARRY sia posto a 0 ed esegue un salto in caso positivo (cioe' se e 0).

Tuttavia e' sempre bene osservare una precauzione quando si esegue un controllo di questo FLAG.

La precauzione e' di assicurarsi che il flag sia nello stato desiderato prima dell' operazione che eventualmente possa modificarlo.

L' istruzione per eseguire questa funzione e' la seguente:

CLC CLear the Carry

Cioe' PULISCI o metti a 0 il flag di CARRY o semplicemente Carry.

### PROGRAMMA 23

```
10 *=828
20 CLC
30 LDA #0
40 SOMMA ADC #1
50 BCC SOMMA
60 STA 1024
70 LDA #1
80 STA 55296
90 RTS
```

Quando gira il programma incrementa progressivamente il contenuto dell' Accumulatore di 1 fino a 255.

L' istruzione ADC # gira gli otto valori 1 in otto 0 e fissa il Carry a 1. Così che quando l' Accumulatore e' visualizzato con l' istruzione STA 1024 si vede il contenuto 0 ( esempio una a commerciale bianca sullo schermo ).

Il 6510 ha una seconda istruzione di controllo per il Carry:

BCS Branch on Carry Set

Questa istruzione controlla che il Carry sia Settato o fissato, per esempio che contenga un 1, e se il controllo da un risultato positivo, esegue un salto. Il seguente programma illustra l' uso di questa istruzione:

#### PROGRAMMA 24

```
10 *=828
30 CLC
40 LDA #0
50 SOMMA1 ADC #1
60 BCS FINE
70 JMP SOMMA1
80 FINE STA 1024
90 LDA #1
100 STA 55296
110 RTS
```

## CORSO DI ASSEMBLER CBM64

Ancora una volta questo programma riempie gli otto bits dell' Accumulatore con dei valori 1, fissa il Carry e termina.

Al termine l' Accumulatore conterra' tutti 0 e per questo la solita a commerciale bianca sara' visualizzata sullo schermo.

Proviamo ora ad addizionare due numeri maggiori del valore 256 che sappiamo essere il massimo esprimibile con un solo byte.

Prima di tutto bisogna calcolare l' MBS e il LBS e per far questo e' necessario passare dal formato decimale a quello esadecimale, al quale per distinguerlo metteremo il prefisso \$.

Per far questo mostriamo un breve procedimento che fa uso della parte comandi Basic del Computer:

|                     |     |             |     |
|---------------------|-----|-------------|-----|
| INT (1157/4096)     | = 0 | I carattere | = 0 |
| INT (1157/256)      | = 4 | II "        | = 4 |
| INT (1157-4x256)/16 | = 8 | III "       | = 8 |
| (1157-4x256-8x16)   | = 5 | IV "        | = 5 |

Per cui sara' :

1157 equivale in esa a 0485

di cui la parte :

MSB = 04

LSB = 85

Per sommare due valori 1157 dobbiamo per prima cosa

addizionare i loro LSB, controllare se c'è un Carry (cioè un riporto) e dopo aggiungere l' MSB tenendo conto della presenza o della mancanza del carry.

```

$85+
$85
--
0A    piu' 1 nel Carry

```

$$16, 10 = \text{Carry} + 0A$$

Dopo si esegue l' addizione sugli MSB

```

$04+
$04
--
08

```

Dopo di che si aggiunge il Carry

$$08 + 01 = 09$$

Nella spiegazione abbiamo ommesso di dire "+ Carry" ed è questa l' operazione che il Flag C esegue per conto del programmatore.

Il Flag infatti è messo a 1 quando l' operazione ha un riporto.

La successiva operazione tiene allora conto di questo riporto e aggiunge 1 alla somma.

Vediamo come si comportano con due brevi esempi i risultati di due somme con diversi valori nel Carry:

Con Carry a 0

$$04 + 04 = 08$$

Con Carry a 1

$$04 + 04 = 09$$

In questo modo la risposta all' esempio precedente e' in esa \$090A o :

$$9 \times 256 + 10 = 2314 \text{ in decimale}$$

Vediamo ora di rifare i calcoli invece che a mano con il computer.

Noi possiamo contare sulla capacita' di manipolazione , da parte del 6510 del Carry, ma non possiamo invece contare sulla sua capacita' di riconoscere quando usarlo.

Tutto il lavoro in doppia precisione e' eseguito prima LSB come durante l' operazione di somma ed il Carry e' immagazzinato per la parte di addizione con il MSB.

Si deve ricordare che quando il 6510 e' usato in comandi di indirizzamento indiretto, questi immagazzina prima il LSB dell' indirizzo e poi il MSB. Questo e' l' ordine usato quando l' indice e' aggiunto al puntatore indirizzo.

Si potrebbe utilizzare questa forma noi stessi quando si immagazzinano NUMERI ( naturalmente distinti dall' indirizzo ).

Per assicurarsi che l' addizione LSB non sia variata dal valore del Carry e' importante far precedere la somma stessa dall' istruzione CLC (Clear Carry).

Prima di tutto dobbiamo calcolare il valore di MSB e LSB in decimale, poiche' entrambi i metodi di immissione dati in memoria lo richiedono.  
Per LSB il suo valore decimale sara':

$$8 \times 16 + 5 = 133$$

mentre per MSB e':

$$0 \times 16 + 4 = 4$$

Ora scriviamo il programma, ma prima di questo introduciamo una nuova istruzione:

NOP No OPeration

Cioe' nessuna operazione.

Quando il 6510 incontra questa istruzione non viene eseguita nessuna operazione per due cicli macchina.

#### PROGRAMMA 25

```
10  *=828
20  CLC
30  CLD
40  LDA #133
50  ADC #133
60  STA 1026
70  LDX #1
80  STX 55298
90  NOP
100 LDA #04
110 ADC #04
120 STA 1024
```

## CORSO DI ASSEMBLER CBM64

```
130 STX 55296
140 RTS
```

Dopo il SYS 828 dovrebbero apparire le lettere I e J sullo schermo.

I passi di questo programma sono specificati e dimostrati nella tabella seguente

| PASSO     | ACCUM. | X | 1026 | 1024 | C |
|-----------|--------|---|------|------|---|
| CLC       | ?      | ? | 0    | 0    | 0 |
| CLD       | ?      | ? | 0    | 0    | 0 |
| LDA #133  | 133    | ? | 0    | 0    | 0 |
| ADC #133  | 10     | ? | 0    | 0    | 0 |
| STA 1026  | 10     | ? | 10   | 0    | 1 |
| LDX #1    | 10     | 1 | 10   | 0    | 1 |
| STX 55298 | 10     | 1 | 10   | 0    | 1 |
| NOP       | 10     | 1 | 10   | 0    | 1 |
| LDA #4    | 4      | 1 | 10   | 0    | 1 |
| ADC #4    | 9      | 1 | 10   | 0    | 0 |
| STA 1024  | 9      | 1 | 10   | 9    | 0 |
| STX 55296 | 9      | 1 | 10   | 9    | 0 |
| RTS       | 9      | 1 | 10   | 9    | 0 |

Come mostra la tavola, all'istruzione ADC #133, viene generato un riporto e il Flag C e' messo a 1 che ha effetto sul seguente ADC.

Altra cosa da notare e' che all'istruzione ADC #4 non c' e' invece nessun riporto e percio' il Carry e' posto a 0.

Per controllare cio' basta rimpiazzare il comando NOP

con CLC che PULIRA' il Flag prima che sia fissato e notare che la risposta data e' errata.

Cio' puo' essere fatto attraverso un comando POKE che immettera' nella locazione 842 il codice per CLC (24)

PROGRAMMA 25

POKE 842,24

Facendo ora girare il programma 24 modificato con 25 saranno visualizzate le lettere:

H J

Infatti il valore di J e' stato calcolato a 266 quindi riportato a 256, il bit di Carry fissato a 1 e' immagazzinato nell' Accumulatore.

### INPUT IN ESADECIMALE

Il sistema esposto in precedenza per la conversione da decimale ad esadecimale puo' sembrare a qualcuno un po' empirico anche se e' sostanzialmente corretto.

Nel programma che mostriamo, invece di inserire numeri decimali immetteremo dei valori espressi in questa nuova notazione e che saranno preceduti dal simbolo del dollaro (\$).

Utilizzando come base il programma 25 otterremo questo nuovo listato:

## CORSO DI ASSEMBLER CBM64

### PROGRAMMA 26

```
10  *=828
20  CLC
30  CLD
40  LDA #$85
50  ADC #$85
60  STA 1026
70  LDX #1
80  STX 55298
90  NOP
100 LDA #$04
110 ADC #$04
120 STA 1024
130 STX 55296
140 RTS
```

Quando questo programma gira, dà lo stesso risultato del programma 24 visto in precedenza.

### Esercizio 13

Usando come input valori esa, sommare \$1807 e \$2AFA. Verificare il programma con addendi e risultati in base 10.

Il 6510 possiede un'istruzione che consente la sottrazione con il Carry. Questa istruzione è:

SBC SuBtract from the accumulator with Carry the data at the specified memory location.

Cioe' sottrai dall' Accumulatore , con riporto il dato contenuto in uno specifico indirizzo di memoria.

Per esempio:

SBC 891

e' un' istruzione che andra' a vedere il valore presente nella locazione di memoria 891 e sottrara' il numero ivi trovato dal valore contenuto nell' accumulatore.

Tuttavia, allo stesso modo in cui si rende necessario preparare il Flag di Carry per l' addizione mettendolo a 0, si rende necessario prepararlo per la sottrazione.

Tuttavia in questo caso sara' necessario invertire il valore del Carry mettendolo a 1 anziche' a 0.

La relativa istruzione e':

SEC SEt the Carry bit to 1

Cioe'metti il bit di Carry a 1

Vediamone ora un' applicazione in un programma che pero' non fara' uso dell' istruzione SBC ma carichera' i valori in modo diretto eseguendo  $4 - 2$ .

## CORSO DI ASSEMBLER CBM64

### PROGRAMMA 27

```
10*=828
20 SEC
30 LDA #4
40 SBC #2
50 STA 1024
60 LDA #1
70 STA 55296
80 RTS
```

Proponiamo un' esercizio.

### Esercizio 14

Scrivere un programma che sottragga 600 da 800 usando l' indirizzamento assoluto. Immagazzinare i dati a partire dalla locazione 890. Visualizzare il risultato in 1034.

### Esercizio 15

Scrivere un programma che sottragga 500 dalla somma eseguita in programma stesso di  $300 + 400$  (tutti i valori in decimale).

Visualizzare la risposta in 1040/1 con l' ordine LSB/MSB

## LA MOLTIPLICAZIONE

Le istruzioni aritmetiche disponibili sul 6510

consentono addizioni e sottrazioni ma non, almeno direttamente, moltiplicazioni.

Questo, cioe' la risoluzione di una moltiplicazione, viene fatta attraverso una serie di somme.

Per esempio  $2 \times 3$  puo' essere espresso come  $2+2+2$  e cio' e' relativamente semplice da programmare.

Il processo da eseguire e' quello di aggiungere all' Accumulatore il valore 2 tre volte e questo richiede che 3 sia fissato in un ciclo che definisce il numero di volte che sara' quindi necessario eseguire la somma.

Ricordiamo che l' Accumulatore all' inizio deve contenere un valore 0.

Vediamo un' applicazione.

#### PROGRAMMA 28

```
10 *=828
20 CLC
30 LDY #3
40 LDA #0
50 SOMMA2  ADC #2
60 DEY
70 BNE SOMMA2
80 STA 1024
90 LDA #1
100 STA 55296
110 RTS
```

Eseguendo il SYS 828 sara' visualizzato una F ( che e' il valore relativo di 6) in 1024.

All' interno di questo programma la chiave e':

```
ADC #2  
DEY  
BNE
```

Questo ciclo che esegue il lavoro e' conosciuto in generale come ALGORITMO della moltiplicazione.

Naturalmente una limitazione di questo semplice algoritmo e' che puo' solo manipolare una risposta con valore non superiore a 255, dopo di che genera un Carry e il valore dell' accumulatore torna a 0.

Si rende necessario anche in questo caso passare al concetto di moltiplicazione in doppia precisione.

Cio' puo' essere ottenuto controllando il Carry dopo ogni somma e se e' stato generato un riporto, aggiungere 1 entro MSB.

Un sistema di controllare l' incremento generato dal ciclo e' di operare con la seguente istruzione:

```
INC INCRement the contents of the specified memory  
location
```

Cioe' incrementa il contenuto di una specifica locazione di memoria.

Il programma 29 mostra l' algoritmo precedentemente visto elaborato per registrare il numero di riporti generati e per incrementare MSB affinche' registri tutto questo.

PROGRAMMA 29

```
10 *=828
```

```
20 LDY #0
```

```
30 STY 905
40 LDY #17
50 LDA #0
60 SOMM16 CLC
70 ADC #16
80 BCC DECY
90 INC 905
100 DECY DEY
110 BNE SOMM16
120 STA 1026
130 LDX #1
140 STX 55298
150 LDA 905
160 STA 1024
170 STX 55296
180 RTS
```

Il programma visualizzera' i caratteri A P.  
Nelle pagine successive vedremo un' altro metodo per le moltiplicazioni.

## LA DIVISIONE

Nello stesso modo che la moltiplicazione e' fatta per somme successive, cosi' la divisione deve essere eseguita per sottrazioni successive.

Questo concetto viene illustrato nel programma 30 in cui 30 e' diviso per 2.

Si sottrae il divisore dal dividendo fino ad ottenere resto 0.

In questo caso l' Accumulatore e' usato per immagazzinare cio' che resta da elaborare, per esempio

## CORSO DI ASSEMBLER CBM64

partendo da 30 ed andando progressivamente verso 0 (30,28,26,24,22, ..4,2,0).

Il registro X e' usato per caricare il divisore in memoria mentre il registro Y memorizza il numero di volte che la sottrazione deve essere eseguita.

### PROGRAMMA 30

```
10 *=828
20 LDY #0
30 LDX #2
40 STX 900
50 LDA #30
60 SOTT2 SEC
70 SBC #2
80 INY
90 CMP 900
100 BCS SOTT2
110 STY 1024
120 LDX #1
130 STX 55296
140 STA 1026
150 STX 55298
160 RTS
```

Dopo il SYS 828 sara' visualizzato il quoziente 15 (come lettera 0) in 1024 ed il resto 0 (come a commerciale) in 1026.

# LEZIONE OTTAVA

## CODICE DECIMALE BINARIO

In aggiunta ai numeri che possono essere rappresentati con la notazione binaria e con quella decimale esiste una forma ibrida o mista appunto la BCD o CODICE DECIMALE BINARIO.

Il BCD forma un ponte fra le due notazioni ed in molti casi facilita grandemente gli output.

Per fortuna il microprocessore puo' manipolare direttamente BCD ed e' in condizioni di operare in questo modo con l'istruzione:

SED SEt Decimal mode.

Questa istruzione fissa automaticamente il Flag D a 1 e percio' le operazioni sono date in BCD.

Quando questo modo di operare non e' piu' necessario allora il flag D e' rimesso a 0 con l'istruzione:

CLD CLear Decimal flag

Questa istruzione, riportando a 0 il flag D, consente di tornare ad operare in binario.

Un semplice programma per sommare 1 a 2 usando il modo BCD e' dato nel programma 31.

Quando gira , questo programma immette una C in 1024.

E' considerato normalmente una buona pratica eseguire un clear sul Flag D dopo ogni operazione di BCD.

L' esempio dato nel 31 e' in effetti identico ad una

## CORSO DI ASSEMBLER CBM64

normale operazione aritmetica con l' eccezione che in BCD il riporto avviene dopo che ogni mezzo byte ( NIBBLE) supera il 9.

Questo e' dimostrato nel programma 32 che aggiunge ancora due 6.

### NOTA

Se il programma 31 fosse ancora in 828 allora il 32 puo' essere POKEGGIATO tramite:

```
POKE831, 6
POKE836, 6
```

### PROGRAMMA 31

```
10 *=828
20 SED
30 CLC
40 LDA #2
50 STA 900
60 LDA #1
70 ADC 900
80 STA 1024
90 LDX #1
100 STX 55296
110 CLD
120 RTS
```

## PROGRAMMA 32

```
10 *=S28
20 SED
30 CLC
40 LDA #6
50 STA 900
60 LDA #6
70 ADC 900
80 STA 1024
90 LDX #1
100 STX 55296
110 CLD
120 RTS
```

Quando gira il programma 32 immette una R bianca in 1024 ( equivalente a \$12 o 16 in decimale ).

Cio' deriva dal fatto che il BCD e' immagazzinato in memoria come NYBBLE, cioe' mezzo Byte.

La lettera R pero' viene come codice del CBM 64 con valore di 18. Come puo' succedere questo?

18 in binario e':

00010010

Tuttavia l' indirizzo di memoria e' immagazzinato in due NYBBLES:

00010010 e' in realta':

0001 cioe' 1 ( in decimale), nel Nybble alto  
e

## CORSO DI ASSEMBLER CBM64

0010 cioè 2 ( in decimale), nel Nybble basso.

per cui il numero rappresentato dai due NYBBLE e':

$1 \times 10 + 2 \times 1 = 12$  (in BCD = 12 esa = 18 decimale).

L' esempio precedente enfatizza il problema che si presenta quando si pensa in modo decimale e si sta lavorando in binario.

In rapporto a quanto abbiamo visto circa il programma precedente dobbiamo trovare una tecnica di manipolazione dei singoli bits entro il Byte.

Per estrarre il Nybble basso da un numero binario e' sufficiente cancellare il nybble alto, ad esempio immettendo tutti 0.

Questo puo' essere fatto con l' istruzione:

AND Esegue un AND logico entro l' Accumulatore.

Un AND e' un operatore logico che confronta due stati logici e produce in uscita, un risultato sulla base del confronto.

Se esaminiamo una porta logica AND come e' usata in un circuito elettronico, si comprende bene anche la funzione assembler AND.

La figura mostra una porta AND con due ingressi A e b e con un' uscita C.

La funzione di questo circuito consiste nel fatto che se entrambi gli ingressi sono a 1 allora anche C sara' a 1. Se invece A o B o tutti e due sono a 0 allora C sara' 0. Cio' e' normalmente espresso in quella che e' conosciuta come TAVOLA DELLA VERITA' ( TRUTH TABLE) che mostriamo di seguito:

| A | B | C |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

### TAVOLA DELLA VERITA' PER AND

#### Esercizio 16

Usando la tavola della verita', calcolare l' output logico ottenuto con i seguenti input:

$$A = 1 \text{ AND } B = 0$$

$$A = 0 \text{ AND } B = 1$$

$$A = 1 \text{ AND } B = 1$$

Quando viene eseguito un AND dal 6510, esso opera su tutti gli 8 bits dell' accumulatore contemporaneamente. Per cui se su 255 viene eseguito un AND di 1 avremo:

## CORSO DI ASSEMBLER CBM64

255 = 11111111

1 = 00000001

cioe':

1 1 1 1 1 1 1 1 Accumulatore

0 0 0 0 0 0 0 1 AND 1

0 0 0 0 0 0 0 1 Risultato

Il risultato crediamo che non abbia bisogno di commenti.

### Esercizio 17

Quale risultato si ottiene eseguendo un AND fra i seguenti due numeri ( base dieci) 149 e 52?

Come abbiamo appena visto nell' esercizio precedente l' istruzione AND puo' essere usata per togliere bits da un numero e potrebbe essere usata per convertire parte del BCD 12 dal programma 32.

Questa istruzione BCD 12 era stata immagazzinata come due NYBBLES in un Byte.

Se il Nybble piu' significativo o MSN puo' essere cambiato in 4 zeri allora il Byte potrebbe essere letto direttamente come Nybble Meno significativo o LSN.

In questo modo il mascheramento di bits puo' essere fatto usando un comando AND.

vediamone il comportamento con BCD 12:

## CORSO DI ASSEMBLER CBM64

```
      0 0 0 1 0 0 1 0      BCD 12
AND   0 0 0 0 1 1 1 1      Binary 15
=     0 0 0 0 0 0 1 0      "      2
```

Eseguendo cioe' l' AND fra BCD ed il numero decimale 15 ( cioe' in binario 00001111) i quattro bits piu' significativi sono stati cancellati ed il numero convertito in LSN ( in questo caso 2 decimale)

In un programma l' istruzione AND puo' essere usata con diversi modi d' indirizzamento. Vediamo un esempio con il modo assoluto:

### PROGRAMMA 33

```
10 *=828
20 LDX #15
30 STX 900
40 LDA #18
50 AND 900
60 STA 1024
70 LDX #1
80 STX 55296
90 RTS
```

Quando gira questo programma, sara' visualizzata una B bianca in 1024.

## CORSO DI ASSEMBLER CBM64

Usando l' indirizzamento immediato ANDIM, il programma 33 puo' essere riscritto come segue:

### PROGRAMMA 34

```
10 *=828
20 LDX #%00001111
30 STX 900
40 LDA #%00010010
50 AND 900
60 STA 1024
70 LDX #1
80 STX 55296
90 RTS
```

### ORA E EOR

Il 6510 usa anche due altri operatori logici uno dei quali consente la funzione OR.

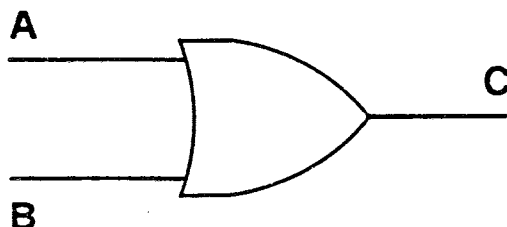
Il codice mnemonico usato in questo manuale per questa funzione e' :

ORA Perform a logical inclusive OR between the Accumulator and the data specified.

Cioe' esegui un OR fra l' Accumulatore ed il dato specificato.

In un circuito elettrico la funzione OR viene

simboleggiata come segue



Il suo modo di operare e' che se un "1" e' presente in A o (OR) in B allora l' uscita C e' messa a "1".

E' un po' il rovescio di AND il quale da come risultato "1" solo se entrambe gli ingressi sono a "1", mentre OR da "0" solo se entrambe gli ingressi sono a "0".

Di seguito la tavola della verita' o TRUTH TABLE.

| A | B | C |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

#### TAVOLA DELLA VERITA' PER OR

Nel programma il comando ORA ha il seguente effetto:

Binario di 149 1 0 0 1 0 1 0 1

Binario di 52 0 0 1 1 0 1 0 0

ORA

Binario di 181 1 0 1 1 0 1 0 1

Inserendo in un programma:

PROGRAMMA 35

```
10 *=828
20 LDA #149
30 ORA #52
40 STA 1024
50 LDX #1
60 STX 55296
70 RTS
```

Allo stesso modo di AND l'istruzione ORA ha la possibilita' di numerosi modi d'indirizzamento.

Il terzo operatore logico e':

EOR Perform a logical Exclusive OR between the accumulator and the data specified.

Questa operazione e' probabilmente la piu' facile da comprendere ed e' illustrata dalla seguente tavola della verita':

| A | B | C |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

## TAVOLA DELLA VERITA' PER EOR

Un sistema di esprimere la funzione e' che l' uscita sara' "1" se l' uno o l' altro degli inputs e' "1" ma non entrambi.

Vediamo di usare questa istruzione con un esempio eseguendo un EOR di 149 con 52 (decimali):

149 in Binario 1 0 0 1 0 1 0 1

52       "       0 0 1 1 0 1 0 0  
EOR

161       "       1 0 1 0 0 0 0 1

Il programma per provare cio' e' il seguente:

PROGRAMMA 36

10 \*=828

## CORSO DI ASSEMBLER CBM64

```
20 LDA #%10010101
30 EOR #%00110100
40 STA 1024
50 LDX #1
60 STX 55296
70 RTS
```

Anche questo operatore logico ha diversi modi di indirizzamento per facilitare l'uso in programmi.

### Esercizio 18

Calcolare i risultati delle seguenti operazioni logiche:

- i) 100 AND 87
- ii) 75 OR 27
- iii) 99 EOR 57
- iv) 94 EOR con il risultato di 100 AND 87.

Tutti i valori sono in decimale.  
Scrivere un programma che verifichi ogni operazione.

### ALTRE FORME DI MANIPOLAZIONE DEI BIT

Esistono altre istruzioni 6510 che consentono di

manipolare bits entro un Byte.

Nell' ultimo esempio che usava BCD, l' istruzione AND era in grado di isolare il LSN dal Byte. Tuttavia non era possibile estrarre il MSN usando la logica disponibile.

Usando uno dei comandi di manipolazione bit che mostreremo cio' diventa ora possibile.

LSR Logical Shift of the specified contents one bit to the Right.

Cioe' esegui uno SHIFT, uno spostamento, di un bit verso destra.

Quando viene eseguito questo comando i bits sono spostati ,TUTTI, di un posto verso destra. L' ultimo bit a destra viene immesso nel Carry e la prima posizione del Byte , che resterebbe vuota, viene riempita con uno 0.

Eseguendo una istruzione LSR sul numero 149 (decimale) avremo il seguente risultato:

149 1 0 0 1 0 1 0 1

= 0 1 0 0 1 0 1 0

con 1 nel Carry

Come per altri comandi anche LSR ha numerosi modi di indirizzamento e l' indirizzo particolare informa il 6510 dove si trova il dato che deve essere traslato.

## CORSO DI ASSEMBLER CBM64.

Così:

LSR A consente una traslazione a destra dei dati in Accumulatore.

LSR 900 come sopra dei dati di indirizzo di memoria 900.

Usiamo il modo Accumulatore di LSR cioè LSR A per immettere e far girare il seguente programma:

### PROGRAMMA 37

```
10 *=828
20 LDA #149
30 LSR A
40 STA 1024
50 LDX #1
60 STX 55296
70 RTS
```

Che visualizza l' equivalente decimale di 74 in 1024.

Usando quattro volte lo spostamento il MSN viene messo al posto del LSN ed i 4 bits a sinistra riempiti con "0".

Cio' consente di isolare MSN in un calcolo BCD.

Tutto questo viene dimostrato nel programma seguente che usa il LSR in modo assoluto.

## PROGRAMMA 38

```

10 *=828
20 LDY #18
30 STY 900
40 LDY #4
50 LOOP LSR 900
60 DEY
70 BNE LOOP
80 LDA 900
90 STA 1024
100 LDX #1
110 STX 55296
120 RTS

```

## Commento e spiegazioni

Decimale 18 = binario 00010010

Nel registro Y c'e' il valore decimale 4

|     |   |                      |          |
|-----|---|----------------------|----------|
|     | I | spostamento a destra | 00001001 |
| II  | " | "                    | 00000100 |
| III | " | "                    | 00000010 |
| IV  | " | "                    | 00000001 |

Al termine nel registro Y c' e' il valore 0, nell' Accumulatore il valore 1 e nella locazione video 1024 appare il carattere A.

## Esercizio 19

Supponiamo che la risposta ad un problema BCD sia 86. Scriviamo allora un programma in codice macchina per decodificare questo e visualizziamo la risposta in decimale in 1024 e 1025.

## CORSO DI ASSEMBLER CBM64

Ricordiamo che il valore BCD 86 equivale al % 10000110.

Un'altra istruzione del set del 6510 e' di muoversi di un bits a sinistra:

ASL Arithmetic Shift Left

Cioe' sposta verso sinistra di un bit.

Anche in questo caso i bits del Byte considerato vengono spostati verso sinistra di una posizione.

La posizione piu' a destra che rimarrebbe vuota viene riempita con uno "0". Il bit piu' a sinistra e' immagazzinato nel Carry.

Eseguendo una istruzione ASL sul numero 149 (decimale) avremo il seguente risultato:

```
149    1 0 0 1 0 1 0 1
        0 0 1 0 1 0 1 0 =
```

con 1 nel Carry

Come per altri comandi anche ASL ha numerosi modi di indirizzamento e l' indirizzo particolare informa il 6510 dove e' il dato che deve essere traslato. Cosi' :

ASL A consente una traslazione a sinistra dei bits in Accumulatore.

LSR 900 come sopra dei dati di indirizzo di memoria 900.

Usiamo il modo accumulatore per provare il seguente esempio:

PROGRAMMA 39

```
10 *=828
20 LDA #149
30 ASL A
40 STA 1024
50 LDX #1
60 STX 55296
70 RTS
```

Decimale 149 = % 10010101

Uno spostamento a sinistra 00101010 equivalente al decimale 42 che, immagazzinato nella locazione 1024, fa apparire una \*.

# LEZIONE NONA

## MOLTIPLICAZIONE BINARIA

Abbiamo visto, relativamente ai programmi precedenti che si puo' eseguire una moltiplicazione usando un processo ripetitivo o RE-ITERATIVO, ma abbiamo anche visto che si tratta di un procedimento lungo e dispendioso.

Esistono pero', in particolare con le istruzioni viste e quelle che vedremo, altri e piu' veloci metodi. Vediamoli.

Vediamo prima di tutto il sistema convenzionale di operare.

Si abbia da moltiplicare 13 x 14. Normalmente si definisce 13 come MOLTIPLICANDO e 14 come MOLTIPLICATORE:

```
13 x
14
--
52
130
---
182
```

In questo formato convenzionale, abbiamo per prima cosa eseguito la moltiplicazione del MOLTIPLICANDO per il

## CORSO DI ASSEMBLER CBM64

digit piu' basso del moltiplicatore ed immagazzina questo come primo prodotto parziale  $4 \times 13 = 52$ .

Successivamente si moltiplica il moltiplicando per il secondo digit del moltiplicatore,  $1 \times 13$ , e dopo si moltiplica questo per 10 per ottenere un secondo prodotto parziale,  $13 \times 10 = 130$ .

In questo modo la somma totale e' la somma delle parti,  $52 + 130 = 182$ .

E' possibile in modo semplice attuare la stessa moltiplicazione usando numeri in formato binario.

Per esempio moltiplicando  $5 \times 7$  in binario:

$5 = 0101$        $7 = 0111$

per cui:

$7 \times 5$  equivale a  $0111 \times 0101$

|                     |         |   |        |
|---------------------|---------|---|--------|
| Prodotto parziale 1 | 0111    | = | 0111   |
| "      "      2     | 0000    | = | 0111   |
| "      "      3     | 011100  | = | 011011 |
| "      "      4     | 0000000 | = | 011011 |
|                     | -----   |   |        |

Risposta                      100011

Cioe'  $32 + 2 + 1 = 35$

In questo caso il processo di moltiplicazione in binario si riduce ad una successiva addizione che segue il movimento a sinistra del moltiplicando.

## MOLTIPLICAZIONE AD 8 BIT

Il diagramma a blocchi relativo a questo processo e' dato nella seguente figura dove:

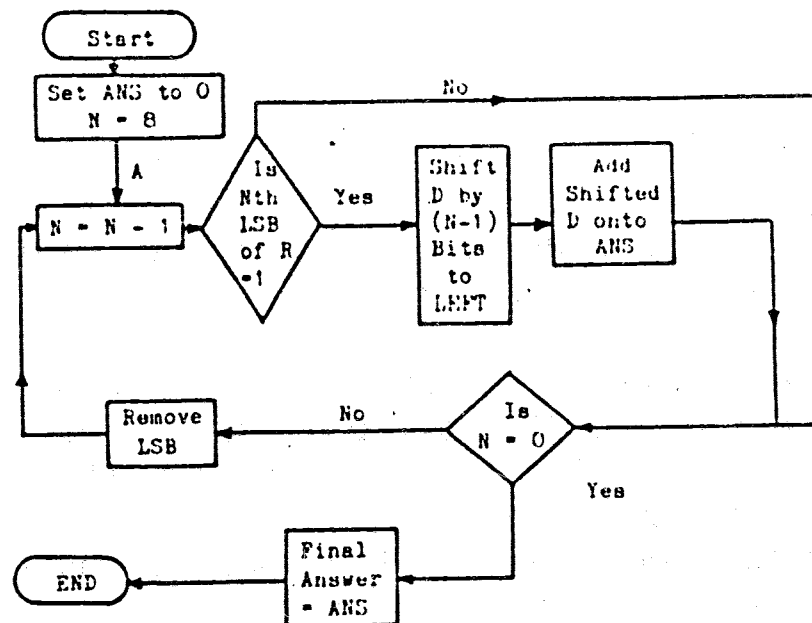
ANS= risposta

D = moltiplicando

R = moltiplicatore

N = numero corrente del bit

LSB= ultimo bit significativo del moltiplicatore.



Attraverso questo diagramma operiamo una semplice moltiplicazione 2 x 2.

Inseriamo ora il concetto in programma:

### PROGRAMMA 40

```

10  *=828
20  LDX #2
30  LDY #22
40  STX 901
50  STY 902

```

## CORSO DI ASSEMBLER CBM64

```
60 LDA #0
70 SALT01 CLC
80 LSR 902
90 BCC SALT02
100 CLC
110 ADC 901
120 SALT02 ASL 901
130 DEY
140 BNE SALT01
150 STA 1024
160 LDX #1
170 STX 55296
180 RTS
```

### Esercizio 20

Riscrivere il precedente programma in maniera che moltiplichi due valori diversi.

Sfortunatamente il programma 40 e' solo una mezza verita' come routine di moltiplicazione ad 8 bit ed opera solo con numeri piccoli sia come moltiplicando che come moltiplicatore.

In una routine completa l'istruzione ASL moltiplica il moltiplicando otto volte per la base. Così con l'ottavo spostamento il bit piu' a destra dovrebbe trovarsi al margine sinistro del registro.

Infatti il secondo bit dovrebbe essere perso dopo il settimo spostamento.

Tuttavia cio' non ha effetto sul risultato complessivo perche' dopo due istruzioni di LSR di 00000010 tutti gli "1" sono stati cancellati e di conseguenza le susseguenti somme parziali saranno uguali a 0.

Se desideriamo usare il programma con numeri maggiori

nella cui risposta finale e' presente un riporto (CARRY), allora le risposte non sarebbero attendibili perche' l' ultimo bit a sinistra era significativo. Fortunatamente l' ultimo bit a sinistra non viene perso nello spazio durante una operazione ASL ma viene inserito nel Carry.

Il problema allora e' di rintracciarlo e di riportarlo nell' MSB della risposta. Cio' puo' essere fatto usando il seguente nuovo comando:

ROL ROTate Left the contents of a specified address.

In questa operazione tutti i bits di uno specifico indirizzo eseguono una rotazione a sinistra ed il bit di Carry viene caricato nella cella del bit piu' a destra mentre il bit piu' a sinistra viene trasferito nel Carry.

7 6 5 4 3 2 1 0 schema del Byte

dopo l' esecuzione di ROL

6 5 4 3 2 1 0 C

nella posizione tenuta precedentemente dallo 0 c' e' ora il contenuto del Carry mentre nel Carry c'e' il contenuto dell' ottavo Bit (cioe' il bit n. 7).

Dato il numero effettivo di bits interessati questa operazione e' chiamata anche ROTAZIONE A 9 BITS. Tuttavia il programma che ne deriva, in particolare per le moltiplicazioni ad 8 bits e' molto piu' complesso e vedremo come si usa con le labels nel capitolo seguente.

L'istruzione appena vista ne porta di conseguenza logica un'altra per la rotazione a destra:

ROR ROTate Right the contents of the specified address.

Cioe' esegui la rotazione a destra di un dato contenuto in uno specifico indirizzo.

Entrambe le istruzioni viste possono essere utilizzate in forme diverse come:

ROL A ROTate Left the contents of the Accumulator.

ROR A ROTate Right the contents of the Accumulator.

Che si spiegano da sole.

E' disponibile un'altra istruzione per la manipolazione dei Bits:

BIT AND specified content's BIT's with accumulator.

Cioe' esegui un AND logico fra il contenuto di un Byte di locazione di memoria data con l'Accumulatore.

Ad esempio l'istruzione BIT 900 esegue un AND logico fra il contenuto dell'Accumulatore e il contenuto della locazione di memoria 900.

Mentre BIT consente la stessa funzione di AND ne differisce in quanto lascia sia l'Accumulatore che la memoria come sono. Sono pero' modificati numerosi FLAGS nel PWS. Vediamo cosa accade:

1) Il flag Z viene settato se il risultato dell'AND e'

zero e viceversa messo a 0 se il risultato e' diverso da zero.

2) Per il flag N invece e': Il bit 7 della locazione che deve essere controllata e' copiato nel Processor Status register.

Questo e' un sistema molto conveniente di controllare quando il contenuto di una particolare locazione sia positiva o negativa senza la necessita' di caricarne il valore entro uno dei due registri.

3) Il Flag V ( che non abbiamo ancora visto in dettaglio) e' il bit 6 del PSR. L'istruzione BIT copia il bit 6 della locazione che deve essere controllata nel bit 6 del PSR. Cio' non e' utile come il Flag N visto prima in quanto il bit 6 normalmente non e' molto importante. Vedremo tuttavia che il Basic lo adopera molto spesso.

Usando queste istruzioni in binario, puo' essere messo in funzione un procedimento analogo a quello visto per la moltiplicazione vista in precedenza.

### DIVISIONE BINARIA A 8 BIT.

Questo procedimento e' analogo a quello visto nella routine di moltiplicazione binaria dato che necessita solo di 8 RE-ITERAZIONI per manipolare un numero di 8 bits.

E' illustrato nel programma seguente dove il dividendo ( nel caso 31) e' immagazzinato nella locazione 900 ed il divisore (2) in 901.

Il registro Y e' usato come contatore di ciclo per

## CORSO DI ASSEMBLER CBM64

assicurarsi che l' algoritmo relativo venga eseguito 8 volte.

Tramite le istruzioni ASL e ROL A il RESTO della divisione e' inserito nell' accumulatore.

### PROGRAMMA 41

```
10  *=528
20  LDX #31
30  STX 900
40  LDX #2
50  STX 901
60  LDY #8
70  LDA #0
80  SALT01 ASL 900
90  ROL A
100 CMP 901
110 BCC SALT02
120 SBC 901
130 INC 900
140 SALT02 DEY
150 BNE SALT01
160 LDX 900
170 STX 1024
180 LDY #1
190 STY 55296
200 STA 1026
210 STY 55298
220 RTS
```

Quando questo programma gira verra' visualizzato il quoziente 15 ( come una 0) in 1024 ed il resto 1 ( come una A) in 1026.

# LEZIONE DECIMA

## LE LABELS

L'uso delle labels consente al programma di dirigersi verso istruzioni con nome senza la necessita' di calcolare salti e relativi indirizzi.

Un termine usuale per Labels e' LABEL SIMBOLICHE perche' le labels stesse sono simboli di locazioni di memoria. Per esempio l'istruzione:

```
BNE LOOP1
```

comunica all'Assembler di costruire un codice macchina che dica al 6510 di saltare ad un'istruzione chiamata ( o con label ) LOOP1.

LOOP1 STA 1024,X crea una label chiamata LOOP1 il cui indirizzo e' lo stesso di quello dell'istruzione STA 1024,X.

Per questo comunque l'inizio di LOOP1 dovrebbe essere immessa come:

```
LOOP1 STA 1024,X
```

Altre convenzioni devono essere osservate quando si usano le labels particolarmente per quanto riguarda gli spazi.

La Label puo' essere lunga da un minimo di un carattere ad un massimo di 10. Si possono anche fare piu' lunghe, ma solo i primi 10 caratteri saranno considerati. Comunque non deve contenere spazi.

## CORSO DI ASSEMBLER CBM64

Anche in questo caso non esiste ragione particolare tranne che questo Assembler quando trova uno spazio dopo la label la considera conclusa.

Per questa stessa ragione la label deve essere seguita da uno spazio e poi da una normale istruzione.

Quando ci si riferisce ad una label in un' istruzione e' necessario solo rimpiazzare l' operando dell' istruzione con la label stessa.

Per semplificare le cose passiamo a vedere come al solito un esempio applicativo del concetto appena esposto.

Il programma seguente usa due cicli ( loops ) chiamati LOOP1 e LOOP2 ed esegue alcuni salti non necessari a scopo dimostrativo.

### PROGRAMMA 41

```
10  *=528
20  LDX #160
30  LDY #1
40  JMP LOOP2
50  LOOP1 LDA #83
60  STA 1183,X
70  DEX
80  BNE LOOP1
90  JMP FINE
100 LOOP2 LDA #90
110 STA 1023,X
120 TYA
130 STA 55296,X
140 DEX
150 BNE LOOP2
160 LDX #120
170 DEY
180 JMP LOOP1
190 FINE RTS
```

Sebbene questo programma esegua dei salti su qualcosa, e' ancora relativamente facile da seguire. Quando il processo di assemblaggio e' terminato il programma risiederà in memoria nello stesso identico formato di un qualsiasi altro programma.

### Esercizio 21

Aggiungere un terzo ciclo LOOP3 al programma precedente. Riscrivere il programma facendo girare per primo il LOOP3 seguito dal LOOP1 e LOOP2. Il LOOP3 dovrebbe far apparire sullo schermo 2 righe di asterischi rossi.

## MEMORY LABELS

In aggiunta alle istruzioni LABEL, l' assembler consente anche la creazione di LABEL come locazioni di memoria. Questa sarà seguita immediatamente dal nome assegnato a quella locazione e, dopo uno spazio, dalla locazione stessa in decimale. Per cui l' istruzione:

LSB 900

informa l' Assembler che, nel resto del programma, la locazione 900 deve riferirsi come LSB. Il seguente programma illustra l' uso di labels di memoria in una somma in doppia precisione che addiziona due numeri a 16 bits.

## CORSO DI ASSEMBLER CBM64

Numero 1 = 2760

" 2 = 948

immessi in LSB1 e MSB1 e LSB2 e MSB2.

La risposta sara' immessa in ANSLSB1 e ANSMSB2

### PROGRAMMA 42

```
10 *           =828
20 LSB1        =900
30 MSB1        =901
40 LSB2        =902
50 MSB2        =903
60 ANSLSB      =904
70 ANSMSB      =905
80 LDA #10
90 STA MSB1
100 LDA #200
110 STA LSB1
120 LDA #3
130 STA MSB2
140 LDA #180
150 STA LSB2
160 CLC
170 ADC LSB1
180 STA ANSLSB
190 STA 1025
200 LDA MSB1
210 ADC MSB2
220 STA ANSMSB
230 STA 1024
240 LDX #2
250 STX 55296
260 STX 55297
270 RTS
```

## CORSO DI ASSEMBLER CBM64

Come abbiamo già visto una riga di programma Assembler può essere composta come segue:

N.LINEA LABEL OPCODE VALORE

Il numero di linea ha la funzione simile che nel Basic.

Della Label ne abbiamo già parlato.

L'opcode o codice operativo non è altro che il codice simbolico di uno dei 56 comandi. LDA è un OPCODE come lo è RTS.

Il valore invece è quel dato che deve seguire il codice mnemonico.

Abbiamo già avuto occasione di vedere che non tutti i comandi o OPCODE necessitano di un valore a seguire.

Soffermiamoci un attimo più approfonditamente sul valore ricordando che deve essere preceduto da un simbolo o prefisso che indicherà in quale forma si vuole operare, vediamo.

Nessun segno davanti al numero indica un valore decimale.

Il segno \$ (dollaro) indica che quello che segue è un valore esadecimale.

Il segno % un valore binario.

Il segno @ (chiocciola o AT o a commerciale) un valore ottale.

Il segno apostrofo (') un valore ASCII.

Cio' quando si intendono i valori impiegati come locazioni di memoria. Nel caso si voglia assegnare un valore immediato il prefisso riferentesi al tipo di numerazione va fatto precedere dal segno (#).

Perciò LDA 1 caricherà nell' Accumulatore il valore contenuto nella locazione di memoria UNO, mentre il comando LDA #1 caricherà nell' Accumulatore il valore

## CORSO DI ASSEMBLER CBM64

1.

LDA 1024 caricherà nell' Accumulatore il valore presente nella locazione (decimale) 1024, mentre LDA #1024 è un comando completamente errato perché il valore immediato può essere SOLO compreso fra 0 e 255 al massimo.

### IL MONITOR

Il programma Assembler fornito su cassetta offre un' altro sistema che potremo definire complementare rispetto a quanto visto fino a questo momento ma che si dimostrerà di non poco aiuto quando avrete preso una certa pratica di questo nuovo modo di operare.

Parliamo del MONITOR per entrare nel quale è sufficiente digitare il comando diretto TIM.

Il comando TIM serve per entrare nel monitor ed abilitarne quindi tutti comandi che elenchiamo in ordine alfabetico e che vedremo in un breve sommario di seguito.

D = DISASSEMBLE  
G = GO  
H = HUNT  
L = LOAD  
M = MEMORY  
R = REGISTER DISPLAY  
S = SAVE  
T = TRANSFER  
X = RITORNO AL BASIC

I comandi sono di un singolo carattere alfabetico

seguiti da uno o piu' parametri se richiesti o se servono.

I parametri possono includere l' indirizzo di partenza o l' indirizzo di partenza e di fine, il codice operativo o OP-CODE, gli operandi, i valori esadecimali, ecc.

I comandi sono eseguiti immediatamente dopo aver premuto il tasto di RETURN.

E' da segnalare che rimane in funzione l' editing tipico del CBM 64, ma non per tutti i comandi si possono utilizzare i cursori per il riposizionamento.

Durante l' immissione del comando e di alcuni parametri se siete incorsi in errore di sintassi, sara' visualizzato un punto interrogativo (?) che segue la posizione dell' errore.

### DISASSEMBLE

Sintassi: D (indirizzo di inizio) (indirizzo di fine)

Questo comando serve per disassemblare programmi, routines o piu' in generale gruppi di codici fra due indirizzi specificati.

Il comando D consente di riconvertire i codici presenti nella memoria del computer nei corrispondenti codici mnemonici del linguaggio Assembler.

Questi verranno presentati su video su 24 righe, nella forma: Numero progressivo, Locazione di memoria, Valori esadecimali, Codice mnemonico. Vediamone un esempio:

```
22 3000 A902    LDA #$02
23 3002 8D1523  STA $2315
```

Se il numero delle informazioni richieste superano la capacita' anzidetta del video la visualizzazione si ferma ed attende che sia premuto un tasto qualsiasi per

## CORSO DI ASSEMBLER CBM64

vedere il gruppo successivo.

Per interrompere la visualizzazione premere il tasto RUN/STOP.

### GO

Sintassi: G (indirizzo)

Serve per mandare in esecuzione un programma partendo da un determinato indirizzo oppure dal valore della locazione contenuta in quel momento nel Program Counter nel caso il parametro venga omissso.

Nell' utilizzare questo comando e' necessario fare molta attenzione perche' se si impiegano dei parametri errati e' facile finire in un LOOP o ciclo infinito, dopo il quale sarebbe necessario spegnere il sistema con la perdita dei dati che possono essere stati memorizzati fino a quel momento.

Ricordare che il parametro deve essere espresso in forma esadecimale.

### HUNT

Sintassi: H (indirizzo di inizio) (indirizzo di fine)  
(dati)

Serve per cercare dei dati in un blocco di memoria specificato dai parametri.

Il comando HUNT localizza ogni selezionato gruppo di caratteri in memoria e li visualizza sullo schermo. I dati devono essere espressi in forma esadecimale e verranno visualizzati sullo schermo le corrispondenti locazioni di memoria in cui si trovano.

Anche in questo caso per fermare anzitempo la funzione

di ricerca e' necessario premere il RUN/STOP.

### LOAD

Sintassi: L "nome del file", (numero della periferica)

Il comando LOAD consente, allo stesso modo che possiamo osservare in un normale programma Basic, di caricare un file di dati o un programma da una determinata periferica nella memoria RAM del CBM 64. Si possono caricare quindi files da nastro o da disco.

Per i files disco, l'indirizzo della prima locazione di memoria RAM entro la quale il file dovra' essere caricato deve essere costituita dai primi due bytes del file.

I files provenienti da cassetta hanno come indirizzo di inizio parte dell' HEADER BLOCK.

Il comando e' composto dalla lettera L, dal nome del file e dal numero di periferica da cui deve essere letto.

Il nome delle file deve essere racchiuso fra virgolette o apici (" ") e naturalmente si deve applicare la sintassi generale del Basic relativa a questa operazione.

Ricordiamo che il numero di periferica per la cassetta e' 01 mentre quella del disco, salvo modifiche hardware e' 08.

### MEMORY

Sintassi: M (indirizzo di partenza) (indirizzo di fine)

Il comando MEMORY, abbreviato con la sola lettera M visualizza il contenuto della memoria dall'indirizzo di partenza specificato nel primo parametro all'indirizzo

di fine incluso nel secondo parametro.

La visualizzazione mostrerà su ogni riga l'indirizzo in esadecimale ed il contenuto, sempre in esadecimale, di 5 bytes di memoria.

Il contenuto della memoria può essere variato scrivendo sopra al valore visualizzato e premendo successivamente il tasto RETURN.

L'indirizzo della prima locazione di memoria esaminata, relativamente al gruppo di 8 Bytes detto, appare alla sinistra della riga.

Se si tenta di modificare i valori contenuti in zone di memoria riservate, come ad esempio i valori contenuti in ROM, allora accanto alla locazione di memoria che non è possibile variare, apparirà un punto interrogativo (?) a segnalare la pratica impossibilità dell'operazione.

### REGISTER

Sintassi: R

Il comando R consente di visualizzare il contenuto dei seguenti registri del microprocessore 6510 il cui significato è già stato visto in precedenza:

|      |   |                   |
|------|---|-------------------|
| ADDR | = | Program Counter   |
| IRQ  | = | Registro di IRQ   |
| SR   | = | Registro di stato |
| AC   | = | Accumulatore      |
| XR   | = | Registro X        |
| YR   | = | Registro Y        |
| SP   | = | Stack Pointer     |

Questo comando è molto utile quando si sta provando un

## CORSO DI ASSEMBLER CBM64

programma perche' consente di osservare se i registri contengono i valori che si desidera.

Si puo' variare molto semplicemente il valore dei singoli registri, dopo aver selezionato R, semplicemente riposizionandosi sopra i valori che appaiono sullo schermo, variandoli e premendo poi il Return.

I registri appena visti verranno automaticamente visualizzati quando si entra per la prima volta nel monitor oppure quando si trova un BREAKPOINT in conseguenza dell' esecuzione di un comando G.

Nel primo caso verra' sempre visualizzato il seguente messaggio:

```
CALL @ 814B
      ADDR IRQ  SR AC XR YR SP
      .;814B EA31 4D 00 00 00 FB
```

### NOTA

Ricordiamo che il contenuto dei registri A, X, Y puo' variare anche in conseguenza delle operazioni eseguite in precedenza.

### SAVE

Sintassi: S "nome del file", (n. della periferica), (indirizzo di inizio), (indirizzo di fine).

Scrivere il contenuto di una data zona di memoria su una particolare periferica che potra' essere disco o cassetta.

I parametri del comando SAVE consistono nel nome del file, nel numero della periferica, 01 per il nastro e 08 per il disco, e negli indirizzi di inizio e fine dati che devono essere in questo caso specificati a differenza di quanto avviene per il simile comando SAVE

## CORSO DI ASSEMBLER CBM64

del Basic.

Gli indirizzi di inizio e fine sono naturalmente indirizzi esadecimali della memoria RAM nella quale e' presente in quel momento il file da salvare.

Inoltre il nome del file non deve superare i 16 caratteri e l'indirizzo iniziale deve essere quello della locazione di memoria in cui comincia il file, mentre quello finale deve essere di un Byte in piu' dell' ultima locazione di memoria che si vuole salvare.

### TRANSFER

Sintassi: T (indirizzo di inizio), (indirizzo di fine), (indirizzo di inizio):

Consente di rilocare un programma o una serie comunque di dati in altra area della memoria disponibile.

Cio' puo' risultare utile quando si desidera espandere un programma o se si vuole usare parte di un programma o di dati senza doverli riscrivere.

Nel comando sono presenti 3 parametri relativi a 3 diversi indirizzi.

I primi due delimitano il blocco di memoria che deve essere trasferito mentre il terzo indica l'indirizzo di inizio della zona in cui il blocco dati deve essere trasferito.

Se il programma da trasferire contiene indirizzi assoluti o WORD TABLE, il trasferimento avverra' lo stesso, ma e' necessario osservare che questi ultimi dati non avranno piu' una logica all'interno delle funzioni di programma.

### RITORNO AL BASIC

Sintassi: X

Serve per uscire dal monitor, che potra' sempre essere riattivato in qualsiasi momento con il comando TIM e riportarsi nell' editor.

## ALTRE INFORMAZIONI SUL BASIC

Quasi tutti gli Assembler consentono delle cosi' dette DIRETTIVE ASSEMBLER o PSEUDO-OPERANDI che sono delle istruzioni particolari e che variano da un linguaggio assemblatore all' altro.

Abbiamo gia' visto l' asterisco (\*) che sta ad indicare l' INDIRIZZO ATTUALE e che abbiamo impiegato all' inizio di ogni programma ma che puo' essere adoperato anche in qualsiasi altro punto del programma sorgente stesso per variare appunto l' indirizzo del PC.

Gli altri 3 pseudo operandi sono WOR, BYT e TXT tutti impiegati per immettere dati, distinti quindi dalle istruzioni, nella memoria del computer attraverso il programma.

WOR immette una parola di 2 Bytes in memoria. Provvede inoltre a separare il numero in Byte di ordine alto e Byte di ordine basso ed immagazzinarlo nel corretto ordine cioe' prima il basso e poi l' alto.

Ad esempio WOR\$7F12 immettera' i Bytes in memoria \$12 e poi \$7F.

Questi dati dovrebbero essere immagazzinati a partire dall' INDIRIZZO ATTUALE.

Supponiamo, per esempio, che in un programma si faccia uso di tavole di valori che inizino a \$9FF0, si potrebbe includere le seguenti linee nel programma da assemblare:

```
200 *=$7FF1  
210 WOR $2E14,$115B,$513A
```

questi sarebbero immagazzinati a partire dalla locazione esa \$7FF1 nella sequenza 14 2E 5B 11 3A 51.

E' da notare che, allo stesso modo che nel comando DATA per il Basic, quando si impiegano piu' valori nello stesso comando WOR, questi andranno separati da virgole.

Lo pseudo codice BYT esegue la stessa funzione vista in precedenza ma per il singolo Byte. Per esempio:

```
BYT $0D
```

immetterà un ritorno carrello all' indirizzo attuale. Con questo pseudo codice si possono anche immettere codici ASCII per le lettere purché siano precedute dall' apostrofo (').

```
1000 BYT 'A
```

immette la lettera A nell' indirizzo attuale.

TXT e' invece piu' utile quando si desideri immettere delle frasi. Deve essere utilizzato immettendo la frase che si vuole immagazzinare fra virgolette.

Si voglia ad esempio immettere la frase PREMI UN TASTO a partire dalla locazione \$9FF0 alla quale si salterà con opportuna subroutine, dovremo procedere in questo modo:

```
200 *=$9FF0  
210 TXT "PREMI UN TASTO"
```

# LEZIONE UNDICESIMA

## COLLEGAMENTO CON IL BASIC

Tutti i programmi in codice macchina sono stati inseriti tramite l' Assembler fino a questo momento.

Tuttavia, come abbiamo visto nell' ultimo capitolo, l' Assembler non e' la sola strada per inserire codici macchina che appunto possono essere caricati sotto forma di POKE come nel seguente esempio.

Per prima cosa carichiamo l' Assembler se non e' piu' in memoria per qualsiasi motivo. Poi digitiamo quanto segue.

### PROGRAMMA 43

```
POKE 828,160
POKE 829,1
POKE 830,162
POKE 831,0
POKE 832,169
POKE 833,90
POKE 834,157
POKE 835,0
POKE 836,4
POKE 837,152
POKE 838,157
POKE 839,0
POKE 840,216
```

## CORSO DI ASSEMBLER CBM64

```
POKE 841,232  
POKE 842,208  
POKE 843,244  
POKE 844,96
```

A differenza di altri Assembler, anche quando il linguaggio e' caricato nella memoria del computer, si possono far girare programmi Basic come quello appena visto.

Ricordiamo tuttavia che e' necessario adottare alcune cautele per non interferire nelle locazioni di memoria dove e' presente il linguaggio Assembler stesso. Ricordiamo inoltre che per resettare il sistema e' sufficiente digitare SYS 64738.

Quando e' stato inserito questo piccolo programma, puo' essere fatto girare con un comando :

```
SYS 828
```

che fara' visualizzare 256 quadri bianchi sullo schermo. Vediamo come si presenta in linguaggio Assembler questo programma.

Per prima carichiamo di nuovo l' Assembler se questi non e' piu' in memoria per un qualsiasi motivo, facciamoci listare il programma a partire dall' indirizzo iniziale cioe' 828.

Digitiamo il comando DISASSEMBLE 828-844 e possiamo vedere di seguito il listato disassemblato del programma inserito in precedenza con in comandi POKE.

### NOTA

Ricordiamo che un effetto simile puo' essere osservato

## CORSO DI ASSEMBLER CBM64

anche utilizzando il comando TIM per accedere al MONITOR.

### PROGRAMMA 44 (ex43)

|   |      |        |              |
|---|------|--------|--------------|
| 0 | 033C | A001   | LDY #\$01    |
| 1 | 033E | A200   | LDX #\$00    |
| 2 | 0340 | A95A   | LDA #\$5A    |
| 3 | 0342 | 9D0004 | STA \$0400,X |
| 4 | 0345 | 98     | TYA          |
| 5 | 0346 | 9D00D8 | STA \$D800,X |
| 6 | 0349 | E8     | INX          |
| 7 | 034A | D0F4   | BNE \$0340   |
| 8 | 034C | 60     | RTS          |

La prima colonna si riferisce ad un numero progressivo di riconoscimento, ma non ha nessun valore pratico. La seconda contiene l'indirizzo di memoria dell'istruzione, la terza i valori posti nella memoria, la quarta i codici mnemonici dei comandi.

Allo stesso modo in cui e' stato possibile inserire un programma senza l'aiuto dell'Assembler e' anche possibile farlo girare direttamente o da un programma Basic.

Come abbiamo visto per farlo girare direttamente e' sufficiente comunicare al Program Counter l'indirizzo di partenza con un SYS all'indirizzo stesso.

Proviamo ora a farlo girare da un programma Basic.

## CORSO DI ASSEMBLER CBM64

### PROGRAMMA 45

```
20000 PRINT "clear"  Far eseguire un Clear di schermo
20010 SYS 828
20020 PRINT "prova"
```

digitiamo quindi:

```
RUN 20000
```

e vediamo che il programma girera' visualizzando i soliti 256 quadri seguiti pero' questa volta dalla scritta prova.

Commentiamo quanto appena scritto.

Linea 20000 Il programma Basic esegue un CLEAR di schermo

Linea 20010 Viene preso il controllo del codice macchina a partire dalla locazione di memoria 828 e vengono eseguite le istruzioni relative fino a quando non si incontri un RTS che fara' ritornare nuovamente il sistema in ambito Basic.

Linea 20020 Il Basic fa stampare il messaggio "prova".

Come abbiamo visto far girare un programma direttamente e' facile ma l' inserimento con i POKE come abbiamo appena visto e' particolarmente noioso, mentre piu' semplice appare la via dei comandi DATA. Vediamone un esempio.

### PROGRAMMA 46

```
1 FOR X = 0 TO 16
2 READ A
3 POKE (528-X),A
4 NEXT X : RESTORE
5 DATA 160, 1, 162, 0, 169, 90, 157, 0, 4
6 DATA 152, 157, 0, 216, 232, 208, 244, 96
7END
```

Questo programma gira come un normale programma Basic con un semplice RUN.

### PROTEZIONE PROGRAMMI

Due comandi del monitor consentono di salvare e di ricaricare poi da nastro o da disco il contenuto di zone di memoria.

Tuttavia abbiamo sempre adoperato il Buffer di cassetta come zona di immagazzinamento programmi per cui, dovendo ora utilizzarle si creerebbero sicuramente problemi di sovrascrittura di dati. Per proteggere i programmi sara' quindi necessario scriverli o spostarli in altra parte della memoria.

Vediamo un po' la memoria del CBM64.

Nel CBM 64 i programmi in Basic sono normalmente immagazzinati fra 2048 (\$0800) e 40595 (\$9FFF), pero' le stringhe generate dal programma sono scritte dal termine della memoria INDIETRO.

Il CBM64 conosce il punto di immagazzinamento di questa informazione perche' quando il computer viene acceso,

## CORSO DI ASSEMBLER CBM64

viene eseguito un controllo della memoria disponibile e le relative informazioni immagazzinate nelle locazioni 55 e 56 di pagina zero.

Se un programma in codice macchina viene scritto al massimo della memoria sicuramente avremo un fenomeno di sovrascrittura delle informazioni in quanto sia un programma Basic che l'Assembler stesso scriveranno o utilizzeranno almeno qualche stringa.

Tuttavia, prima di immagazzinare le stringhe con il sistema DAL MASSIMO DELLA MEMORIA INDIETRO il Basic controlla le locazioni di memoria 55 e 56 per sapere dove scrivere.

Sara' quindi sufficiente proteggere una parte della memoria abbassando i valori dei due puntatori.

Immediatamente dopo l'accensione l'indirizzo 40960 (\$A000) viene immagazzinato in 55 e 56 con il solito sistema di:

55 LSB cioe' 00

56 MSB cioe' A0

Per proteggere una zona di 100 bytes questo indirizzo dovrebbe essere abbassato a 40860 (\$9F9C).

Per far cio' ci sono numerose strade. In modo diretto potremo avere:

POKE 55, 156 (in esa e' 9C)

POKE 56, 159 (in esa e' 9F)

seguiti da :

CLR

necesario per informare il Sistema Operativo di aggiustare tutti gli altri puntatori di conseguenza.

L' abbassamento della memoria utile per il Basic puo' essere fatto anche in modo programma immettendo le stesse istruzioni:

```
10 POKE 55,156 : POKE 56,159 : CLR
```

Altro metodo puo' essere quello di utilizzare il Monitor.

Si trovano le locazioni di memoria 55 e 56 che corrispondono ai valori esa :

55 = \$ 37

56 = \$ 38

e li si scrivono i nuovi valori , nel nostro caso 9C e 9F.

Ci sono tuttavia dei problemi o possono esserci, nell' abbassare i puntatori della memoria.

Infatti gli indirizzi possono essere stati abbassati da un precedente programma o per altri motivi.

Per questo motivo sara' bene controllare i valori contenuti nei puntatori.

Il seguente programma Basic mostra come abbassare i puntatori e quindi la memoria di 200 bytes sotto controllo

## CORSO DI ASSEMBLER CBM64

### PROGRAMMA 47

```
100 MT =256*PEEK(56) +PEEK(55): REM TROVA IL VECCHIO  
INDIRIZZO  
110 MN = MT -200 : REM CALCOLA NUOVO INDIRIZZO  
120 MH = INT(MN/256) :REM PER HI-BYTE  
130 ML = MN - 256* MH : REM PER LO-BYTE  
140 POKE 56, MH : POKE 55,ML : CLR
```

# LEZIONE DODICESIMA

## I COLORI NEL CBM 64

Una delle maggiori capacita' del CBM64 e' quella di visualizzare i colori.

Cio' e' disponibile e facilmente usabile sia operando in Basic che in codice macchina.

Il seguente programma mostra una combinazione di colori schermo/bordo.

### PROGRAMMA 48

```
10 *=828
20 LDY #15
30 LDX #15
40 VIDEO STY $D021
50 BORDO STX $D020
60 DEX
70 BPL BORDO
80 DEY
90 BPL VIDEO
100 RTS
```

Sfortunatamente questo programma gira in 2600 cicli pari a 1300 microsecondi e diviene percio' non percettibile dai nostri occhi.

Per poterlo osservare sara' quindi necessario ricorrere a dei cicli di ritardo.

## CORSO DI ASSEMBLER CBM64

Nei precedenti capitoli abbiamo visto molte cose sia sui cicli sia sui ritardi.

Ricordiamo che agli indirizzi di memoria in pagina zero 160, 161 e 162 (\$A0, A1 e A2) esiste un orologio (JIFFY CLOCK) che funziona come un contatore binario.

Il byte di indirizzo 162 (\$A2) viene incrementato di 1 ogni sessantesimo di secondo. Quando arriva a 256 scarica 1 sul byte 161 che a sua volta si incrementa fino ad arrivare a 256 dopo di che fara' scattare il contatore di locazione 160 di uno.

Tutto cio' ci consente di manipolare ritardi di qualsiasi misura come nel programma seguente:

### PROGRAMMA 49

```
10 *=828
20 LDY #15
30 BORDO STY $D020
40 LDX #15
50 VIDEO STX $D021
60 LDA #246
70 STA 162
80 LOOP LDA 162
90 BMI LOOP
100 DEX
110 BNE VIDEO
120 DEY
130 BNE BORDO
140 RTS
```

Anche sui singoli caratteri sullo schermo possono essere controllati i codici di colore dello schermo, come del

resto abbiamo fatto nel nostro ultimo programma. Se all' indirizzo di memoria 55296 (\$D800) c'e' un "2" allora la locazione di memoria 1024, cioe' la prima locazione a sinistra in alto sullo schermo, sara' rossa. O meglio il carattere che verra' visualizzato in quella locazione di memoria sara' rosso.

Ecco la tabella completa dei colori:

| CODICE | COLORE      |
|--------|-------------|
| 0      | Black       |
| 1      | White       |
| 2      | Red         |
| 3      | Cyan        |
| 4      | Purple      |
| 5      | Green       |
| 6      | Blue        |
| 7      | Yellow      |
| 8      | Orange      |
| 9      | Brown       |
| 10     | Light red   |
| 11     | Dark grey   |
| 12     | Mid grey    |
| 13     | Light green |
| 14     | Light blue  |
| 15     | Light grey  |

Il programma 50 mostra come il colore dei blocchi di schermo possa essere definito per mezzo dei codici di colore dello schermo stesso.

## CORSO DI ASSEMBLER CBM64

### PROGRAMMA 50

```
10 *=828
20 LDX #200
30 LOOP LDA #8
40 STA $D7FF,X
50 LDA #160
60 STA 1023,X
70 DEX
80 BNE LOOP
90 RTS
```

### NOTA

Ricordiamo che a proposito della grafica la EVM COMPUTERS ha messo a punto un corso apposito che si chiama appunto:

### CORSO DI GRAFICA PER CBM 64

contenente anche una cassetta con programmi e utility e che puo' essere reperito in libreria oppure direttamente alla EVM.

# LEZIONE TREDICESIMA

## LE ROUTINES DEL SISTEMA OPERATIVO

Per utilizzare il 6510 il nostro computer ha immagazzinato in ROM (Read Only Memory) una serie di routines di controllo.

Queste routines consentono al CBM 64 di riconoscere ed utilizzare i comandi Basic che fanno parte di un programma, tutti gli input e gli output, intesi qui nel senso di ingresso e uscita dati, e tutte le procedure comunque necessarie al suo funzionamento.

Le ROM che manipolano tutti i comandi Basic sono localizzate in memoria fra gli indirizzi \$A000 e \$BFFF. Le ROM che si occupano di tutte le altre routines del Sistema Operativo del computer, chiamate KERNAL dalla Commodore sono fra \$E000 e \$FFFF.

Abbiamo quindi una suddivisione, abbastanza elastica da considerare fra i due grandi blocchi di istruzione:

### IL BASIC

### IL SISTEMA OPERATIVO

In aggiunta alle locazioni in ROM sia il Basic che il S.O. utilizzano parte della memoria RAM di indirizzi fra \$0000 e \$03FF, cioè le prime quattro pagine.

In particolare come abbiamo già detto in precedenza

## CORSO DI ASSEMBLER CBM64

viene largamente utilizzata la pagina zero.

Buona parte di questo utilizzo e' per l'immagazzinamento di dati transienti, cioe' temporanei o che si modificano in continuazione come per esempio abbiamo visto il JIFFY CLOCK, che ci dice anche da quanto tempo e' stato acceso il computer.

Altre zone RAM sono usate per dati con maggiore stabilita' o utilizzate come le locazioni da 43 a 56 (\$2B fino a \$38) che ci indicano l' area di memoria usata dai programmi Basic e le relative aree di dati.

Alcune di queste indicazioni si servono di due o tre Bytes mentre altre, come per esempio il Buffer di cassetta, che viene utilizzata durante il trasferimento di dati da e per la cassetta, sono di dimensioni maggiori.

Infatti il Buffer di cassetta occupa ben 192 Bytes.

L' aspetto piu' difficile nell' uso delle routines inserite (BUILT-IN SUBROUTINES) e' di conoscere da che parte provengono a loro le informazioni e dove esse depositano i dati che producono. Cio' e' particolarmente vero per quanto riguarda le routines che appartengono al gruppo dell' Interprete Basic.

Al termine di questo manuale si trovano delle tavole, abbastanza complete, che descrivono gli indirizzi e le azioni delle routines Commodore. Tuttavia molte funzioni saranno esemplificate e commentate nel presente e nei successivi capitoli.

Per prima cosa diamo un' occhiata ai contenuti dell' Accumulatore durante l' uso di una subroutine kernal.

Nei primi capitoli abbiamo visto l' Accumulatore durante l' uso di un comando STA per spostare una copia dello stesso in una locazione di memoria, es. STA 1024.

Un sistema semplice e piu' facile e' quello di usare una Kernal che si chiama CHROUT di indirizzo 65490 (\$FFD2). Questa routine consente di visualizzare il contenuto

## CORSO DI ASSEMBLER CBM64

dell' Accumulatore a partire dall' attuale posizione del cursore.

Vediamone un' applicazione:

### PROGRAMMA 51

```
10 *=828
20 LDA #42
30 STA 1524  Immagazzina il contenuto dell' Accumulatore
              nel mezzo dello schermo(1542).
40 LDX #1
50 STX 55796
60 JSR $FFD2 Salta alla routine CHROUT.
70 RTS
```

Dopo aver assemblato il programma con il comando Assemble e fatto girare con SYS 828 (RETURN) saranno visualizzati due asterischi.

Nel mezzo dello schermo un asterisco bianco che e' stato immesso direttamente.

Avremo inoltre un' altro asterisco nella riga seguente a quella del SYS 828.

Questo e' il risultato dell' uso della routine CHROUT. Notiamo che questa subroutine non ha specificato ne' la posizione ne' il colore.

I due maggiori vantaggi di questa subroutine sono che per prima cosa localizza automaticamente la posizione del cursore e la incrementa, sempre automaticamente , tutte le volte che la routine stessa e' chiamata. Secondo viene immagazzinato l' attuale colore nell' appropriata posizione RAM colore.

## CORSO DI ASSEMBLER CBM64

Naturalmente se si desidera si puo' diversamente fissare, tramite il programma che scriveremo, una diversa posizione del cursore e un colore diverso.

Fissare un' altro colore e' relativamente facile. Infatti il colore attuale (del cursore) e' localizzato in 646 (\$0286) con i soliti valori: 0 per il nero, 1 per il bianco , ecc.

In questo modo basta immettere il colore desiderato ed e' tutto cio' che e' necessario fare.

Il posizionamento del cursore e' un po' piu' complesso. Fortunatamente una routine, chiamata giustamente PLOT di indirizzo 65520 (\$FFF0) esegue la maggior parte del lavoro.

PLOT puo' leggere o fissare l' attuale posizione del cursore usando i registri X e Y.

Se facciamo entrare questa routine con il flag di Carry settato ( cioe' a 1), allora PLOT riporterà l' attuale posizione del cursore nei registri X e Y, dove X conterra' il numero della riga (da 1 a 24) e Y il numero della colonna ( da 0 a 39).

Se questa routine e' inserita con il flag di Carry CLEAR (cioe' a 0), allora il valore che e' stato immagazzinato in X e Y sara' usato per posizionare il cursore.

Vediamo nel seguente esempio come si puo' usare la subroutine PLOT per immettere un asterisco giallo all' inizio della decima linea di schermo.

### PROGRAMMA 52

```
10  *=628
20  CLC          Esegui il CLEAR.
30  LDX #9       Carica 9 in X (per la decima linea).
40  LDY #0       Carica 0 in Y (per la prima colonna).
```

## CORSO DI ASSEMBLER CBM64

```
50 JSR $FFFF0 Vai a PLOT per posizionare il cursore.
60 LDA #7      Carica il valore 7 per il colore giallo.
70 STA 646     Immetti il contenuto di A (7) come colore
               corrente in 646.
80 LDA #42     Carica l'asterisco.
90 JSR $FFD2   Salta a CHROUT per stampa.
100 RTS
```

Se avessimo desiderato inserire il nostro asterisco nella diciottesima posizione della decima linea avremmo dovuto immettere il valore 17 nell'istruzione LDY (la prima) avendo così il seguente programma:

### PROGRAMMA 53

```
10 *=828
20 CLC
30 LDX #9
40 LDY #17
50 JSR $FFFF0
60 LDA #7
70 STA 646
80 LDA #42
90 JSR $FFD2
100 RTS
```

Per illustrare come la routine CHROUT manovra il cursore in modo tale che sia spostato alla posizione successiva dopo ogni chiamata di routine, proviamo a modificare il programma 53 con:

## CORSO DI ASSEMBLER CBM64

```
LDY #4  
JSR $FFD2  
DEY
```

per cui il nuovo programma sara':

### PROGRAMMA 53/B

```
10  *=828  
20  CLC  
30  LDX #9  
40  LDY #17  
50  JSR $FFFF0  
60  LDA #7  
70  STA 646  
80  LDA #42  
90  LDY #4  
100 LOOP JSR $FFD2  
110 DEY  
120 BNE LOOP  
130 RTS
```

Quando gira il programma 53B visualizzera' quattro asterischi gialli nella linea 10 rispettivamente nelle colonne 18, 19, 20 e 21.

Notare che non e' stato necessario ricaricare l' Accumulatore con il valore 42 tutte le volte che e' entrato in funzione il ciclo (LOOP)perche' la CHROUT non ne altera il valore.

E' anche chiaro che il registro Y non e' stato cambiato. In caso contrario infatti il conteggio dei 4 asterischi non avrebbe funzionato.

## CORSO DI ASSEMBLER CBM64

Come abbiamo detto infatti la routine CHROUT non varia ne A, Y o X. E questa e' un' ottima qualita' di questa utilissima routine.

Purtroppo non tutte le routines di inserimento si comportano in questo modo e spesso sara' necessario tenere in considerazione la possibilita' che invece i registri, uno o piu' di uno, siano cambiati durante il loro funzionamento.

Molti programmi Basic usano il comando GET, che accetta un singolo Byte di ingresso entro il Buffer di tastiera. Prende in pratica un carattere per volta.

Per questa funzione il GET usa una delle routines Kernal chiamata GETIN di indirizzo 65508 (\$FFE4).

Quando entra in funzione questa routine o quando viene chiamata, GETIN rintraccia un carattere dalla coda di tastiera e lo immette, come valore ASCII, nell' Accumulatore.

Se la coda di tastiera (KEYBOARD QUEUE) e' vuota, GETIN non attende, ma riporta un valore zero.

Il programma 54 mostra una operazione con GETIN:

### PROGRAMMA 54

```
10 *=828
20 LDA #0
30 LOOP JSR $FFE4 !GETIN
40 BEQ LOOP
50 JSR $FFD2 !CHROUT
60 RTS
```

Quando gira questo programma fissa il ciclo:

```
LOOP JSR $FFE4  
BEQ LOOP
```

che resta in attesa di un carattere.

Quando viene inviato un dato, il programma passa attraverso l'istruzione BEQ ed esegue il resto del programma stesso, visualizzando il carattere ottenuto per mezzo di GETIN.

Notare che e' stato scelto di usare la subroutine CHROUT per immettere il risultato nello schermo invece di porcelo direttamente.

Sia GETIN che CHROUT operano con il codice ASCII invece del codice schermo CBM64.

Ricordiamo che solo nel caso dei numeri i due codici corrispondono, altrimenti operando in un modo o nell'altro dovremo ricercare i relativi valori nelle tavole per visualizzare gli stessi risultati.

# LEZIONE QUATTORDICESIMA

## ROUTINES DEL SISTEMA

La routine CHRIN di indirizzo 65487 (\$FFCF) e' una alternativa a GETIN.

Quando si esegue un input da tastiera, la sua azione e' simile a quella del comando INPUT del Basic.

I caratteri che erano stati immessi sono immagazzinati nel buffer del Basic che inizia da 512 (\$0200). Vengono inoltre accettati tutti i comandi di edit compresi gli INSERT ed i DELETE.

Tuttavia non abbiamo la necessita' di organizzare la ricerca di questi caratteri nel Buffer del Basic, perche' i caratteri stessi saranno restituiti in sequenza dopo ogni salto o chiamata di CHRIN.

Non e' neppure necessario organizzare la visualizzazione di questi caratteri perche' anche questa parte del lavoro e' organizzata da CHRIN.

Tutto questo lavoro puo' essere quindi sintetizzato in un breve programma:

PROGRAMMA 55

```
JSR $FFCF  
RTS
```

E' preferibile pero' l'impiego della routine GETIN

## CORSO DI ASSEMBLER CBM64

perche' funzionando questa come un GET permette il controllo del carattere immesso.

La routine GETIN puo' essere usata per mettere a punto una vostra routine di INPUT.

Si potrebbe usare GETIN per immettere un carattere per volta, eseguire il controllo di entrata per un certo numero di caratteri oppure per l' inserimento di un certo carattere di termine che non debba essere necessariamente un RETURN .

Si potrebbe anche utilizzare la routine per controllare ogni carattere immesso attraverso l' attuale comando di INPUT e dare una segnalazione di ATTENZIONE se e' un carattere non valido.

Il seguente programma mostra un esempio di quanto detto con controllo di inserimento di una virgola (ASCII 44):

### PROGRAMMA 56

```
10  *=828
20  LDX #44          Immagazzina la virgola in 900.
30  STX 900
40  LOOP JSR $FFE4   Uso di GETIN.
50  BEQ LOOP        Attesa per Input.
60  JSR $FFD2        Visualizza A sullo schermo.
70  CMP 900         Controllo per virgola.
80  BNE LOOP        Se non presente torna indietro.
90  LDA #13
100 JSR $FFD2
110 RTS
```

Questo programma simula una routine di INPUT che termina con una virgola invece che con un RETURN.

## CORSO DI ASSEMBLER CBM64

Per usare un carattere di termine diverso dalla virgola, basta cambiare l'operando della prima istruzione in modo tale che il valore corrispondente al carattere che si desidera sia immagazzinato alla locazione 900.

### Esercizio 22

Modificare il programma 56 in modo tale che sia accettato un input che termini con uno spazio. Usare un comando POKE per ottenere questa variazione.

Come abbiamo detto in precedenza, uno dei maggiori problemi che si possono incontrare nell'uso di queste routine e' che esse fanno, di solito, un largo uso sia del 6510 che dei relativi registri o flags.

Per questo, dopo aver eseguito un'istruzione JSR non sara' ragionevole, al ritorno, pensare o supporre che non sia successo niente nei registri del 6510.

Per esemplificare quanto detto, osserviamo il seguente programma che e' stato messo a punto per eseguire un input di una stringa di 4 caratteri da tastiera.

Per prima cosa viene fissato un contatore di ciclo con valore 4 nel registro X. Sono usate successivamente le routines GETIN e CHROUT.

Di ritorno da queste routines e' decrementato X e controllato se il flag Z e' settato.

## CORSO DI ASSEMBLER CBM64

### PROGRAMMA 57

```
10 *=828
20 LDX #4
30 LOOP JSR $FFE4
40 BEQ LOOP
50 JSR $FFD2
60 DEX
70 BNE LOOP
80 RTS
```

Tuttavia quando questo programma gira, viene eseguito un RTS dopo che un solo carattere e' stato immesso.

Cio' suggerisce che una delle subroutines stia usando il registro X.

Per verificare cio' possiamo mettere il programma in modo che stampi il contenuto del registro X ai vari stadi di passaggio.

Questa operazione e' fatta nel programma seguente in cui il contenuto del registro X e' esaminato immediatamente dopo il ritorno da subroutine (RTS).

### PROGRAMMA 58

```
10 *=828
20 LDX #4
30 LOOP JSR $FFE4
40 BEQ LOOP
50 STX 1024
60 LDY #1
70 STY 55296
80 JSR $FFD2
```

## CORSO DI ASSEMBLER CBM64

```
90 STX 1026
100 LDY #1
110 STY 55298
120 DEX
130 BNE LOOP
140 RTS
```

Quando gira, questo programma si comporta come il precedente stampando il carattere in INPUT ma visualizza anche 2 A alle locazioni 1024 e 1026.

Poiche' la prima A e' stampata immediatamente dopo l'uscita dalla routine di GETIN (JSR \$FFE4) e' chiaro che GETIN modifica il registro X.

Come abbiamo appena scoperto CHROUT non esegue modifiche in questo caso.

Infatti poiche' il registro X viene messo a 1 da GETIN ecco spiegato il comportamento delle istruzioni DEX/BNE che ordinano di lasciare il programma dopo l'esecuzione dl primo ciclo.

Per superare questo problema e' necessario che il valore del registro X sia immagazinato da qualche parte prima di accedere alla subroutine e ripristinato prima che si passi a decrementarlo.

Il programma 59 mostra questo processo in cui X e' temporaneamente immagazzinato nella locazione 900 durante l'esecuzione della subroutine.

### PROGRAMMA 59

```
10 *=828
20 LDX #4
30 SALVAX STX 900
```

## CORSO DI ASSEMBLER CBM64

```
40 GET JSR 65508 ! $FFE4 GETIN
50 BEQ GET
60 JSR $FFD2 !CHROUT
70 LDX 900
80 DEX
90 BNE SALVAX
100 RTS
```

L'uso di questa tecnica se da una parte risolve il problema, dall'altra e' particolarmente laboriosa. Per fortuna il 6510 e' in grado di eseguire automaticamente parte di queste operazioni.

### LO STACK

Lo stack (S) e' un blocco di memoria, sul CBM64 localizzabile da 511 (\$01FF) INDIETRO fino a 256 (\$0100) che e' in grado di manipolare 255 Bytes.

E' usato per un trasferimento rapido dei dati che vengono immessi a partire dalla locazione 511 all'INDIETRO, mentre l'indirizzo della prima locazione libera viene immagazzinato nello STACK POINTER (SP).

Quando qualcosa deve essere ritrovato dallo STACK, solo l'ultima registrazione e' accessibile.

Di norma viene usata un'analogia con una pila di piatti nella quale solo l'ultimo piatto della pila e' accessibile.

Il termine tecnico per questo procedimento e' :

### TECNICA LIFO

## CORSO DI ASSEMBLER CBM64

LIFO vuol dire LAST-IN FIRST-OUT, cioè che l' ultimo dato inserito e' quello che esce per primo.

Una delle funzioni dello Stack e' di ricordare gli indirizzi durante i salti a subroutines, cosa che viene fatta automaticamente.

Quando il 6510 trova un' istruzione come:

```
JSR 50000
```

deve per prima cosa tenere a mente dove e' la prossima istruzione in modo tale che possa trovare il suo nuovo indirizzo dopo che la subroutine e' stata eseguita e quindi piazza 50000 nel Program Counter.

La procedura e' esaminata qui di seguito con un segmento di programma , il 60, proveniente dal programma 59

### PROGRAMMA 60

PASSO 1. 828 (\$033C) STX 900 (\$0384)

PASSO 2. 831 (\$033F) JSR 65508 (\$FFE4)

PASSO 3. 834 (\$0342) BEQ 251

### ISTRUZIONE STX 900

PASSO 1.

I) Calcola l'indirizzo della prossima istruzione(831)

## CORSO DI ASSEMBLER CBM64

- II) Metti l'indirizzo nel PC.
- III) Esegui l'istruzione STX 900.
- IV) Ricava l' indirizzo per la prossima istruzione del PC e che sara' \$033F.

### PASSO 2.

- V) Vai a cercare l'indirizzo della prossima istruzione a \$033F.  
L' istruzione da eseguire e' JSR \$FFE4.
- VI) Ripristina l' indirizzo per la prossima istruzione in programma, \$0342, ed immettilo nello Stack.

### NOTA

L' immissione nello STACK sara' fatta a partire dalla prima locazione libera dello STACK. Supponiamo che sia 511:

Indirizzo da immettere \$0342

509

510 \$03 (MSB)

511 \$42 (LSB)

- VII) Registra che la prima locazione disponibile nell' area Stack e' la 509. Lo Stack Pointer sara' quindi a 509.

VIII) Carica \$FFE4 nel PC.

IX) Esegui il salto a \$FFE4.

X) Esegui la subroutine prima di trovare RTS.

XI) Vai allo Stack Pointer per trovare l'ultimo dato. Poiche' SP e' = 509 i dati saranno a 510 e 511.

XII) Estrai i dati da 510 e 511 (\$0342) caricali nel PC, esegui il reset dello SP a 511.

XIII) Salta a \$0342.

PASSO 3.

XIV) Vai a cercare la prossima istruzione e prosegui con il programma.

Non e' molto semplice.

Fortunatamente le operazioni dello stack relative alla memorizzazione di indirizzi durante l'esecuzione di subroutines e' automatica lasciando al programmatore il tempo ed il modo di occuparsi d'altro.

Tuttavia, come abbiamo visto a proposito delle routines di inserimento, lo Stack non puo' immagazzinare automaticamente il contenuto di tutti i registri, ma deve essere programmato per questo.

Esistono solo due istruzioni per immagazzinare i dati dei registri, nessuna delle quali purtroppo per i due registri che dovranno quindi essere salvati tramite il passaggio attraverso l'Accumulatore.

## CORSO DI ASSEMBLER CBM64

PHA Push contents of Accumulator onto stack

Cioe' immetti il contenuto dell' Accumulatore nello Stack.

contenuto che potra' essere ricercato con l' istruzione:

PLA Pull top of stack into Accumulator.

Usando queste istruzioni, il programma 57 puo' essere riscritto per trasferire il registro X entro lo Stack e ritrovarlo quando necessita.

### PROGRAMMA 61

```
10  *=828
20  LDX #4
30  SALVAX TXA
40  PHA
50  GET JSR $FFE4
60  BEQ GET
70  JSR $FFD2
80  PLA
90  TAX
100 DEX
110 BNE SALVAX
120 RTS
```

Quando questo programma gira, accettera' quattro caratteri in INPUT e li visualizzera' sullo schermo.

Altre due istruzioni che consentono di salvare e ritrovare dati sullo Stack sono:

PHP Push Processor status register on stack

Per caricare lo SR nell' area di Stack, e:

PLP Pull stack to Processor status register

Per rintracciare i dati.

Nel programma 61 lo SR non era stato salvato nell' area di Stack, perche' prima di controllare il flag Z con BNE, l' istruzione DEX lo aveva resettato.

Tuttavia in altre circostanze puo' rendersi necessario salvare il contenuto di SR.

### Esercizio 23

Riscrivere il programma 61 in modo da salvare SR nello Stack prima della subroutine e ritrovarlo dopo l' esecuzione della stessa.

Quando si usa lo stack, la maggiore preoccupazione deve essere quella di controllare l' ordine di entrata e di uscita dei dati ricordandosi che il metodo di ricerca e salvataggio e' il LIFO.

## SCHEMA DI FUNZIONAMENTO

In genere si usa questo sistema di successione:

|    |                                 |                          |
|----|---------------------------------|--------------------------|
| 1  | PHA                             | SALVA A                  |
| 2  | PHP                             | SALVA SR                 |
| 3  | TXA                             | TRASFERISCI X IN A       |
| 4  | PHA                             | SALVA IL VALORE DI X     |
| 5  | TYA                             | TRASFERISCI Y IN A       |
| 6  | PHA                             | SALVA IL VALORE DI Y     |
| 7  |                                 |                          |
| 8  | -----PROGRAMMA DA ESEGUIRE----- |                          |
| 9  |                                 |                          |
| 10 | PLA                             | RICARICA IN ACCUMULATORE |
| 11 | TAY                             | TRASFERISCE IN Y         |
| 12 | PLA                             | RICEVE IN ACCUMULATORE   |
| 13 | TAX                             | TRASFERISCE IN X         |
| 14 | PLP                             | RICARICA SR              |
| 15 | PLA                             | RICARICA ACCUMULATORE    |

# LEZIONE QUINDICESIMA

## INTERRUPTS, NUMERI E VARIABILI

Quando si esegue un lavoro, non si desidera essere interrotti fino a quando non sia stato terminato. Anche il 6510 possiamo dire che si comporta allo stesso modo.

Durante l' esecuzione di un programma il microprocessore ha il controllo dell' Accumulatore, dei registri X e Y e tutti i flags sono appropriatamente settati.

Così all' atto dell' interruzione e' necessario che tutti i registri siano salvati, normalmente nell' area STACK, e dopo l' interruzione, ritrovati e ripristinati. L' interruzione o INTERRUPT e' infatti solo una subroutine che interrompe il lavoro del 6510 quando si desidera.

Questo consente di affermare che l' interrupt e' generato FUORI dal sistema di funzionamento del 6510 sia da un a periferica esterna che, per esempio da tastiera.

La manipolazione dell' interrupt deve essere preparata da programma perche' in casi particolari non sono consentite interruzioni.

Se, per esempio, un' altra periferica sta inviando dati alla memoria, allora' viene messa in funzione una procedura di HAND-SHAKING fra le due macchine.

Questa procedura , in termini semplici si comporta in questo modo.

C' e' uno scambio di messaggi lungo la linea di trasmissione, del tipo:

"Sono pronto per inviare dati.

Sei pronto per ricevere?"

"Si"

"Questi sono i dati..... Fine dati"

"Grazie, OK"

Se avviene un Interrupt, e' probabile che i dati siano troncati e quindi senza nessun valore.

Durante questi periodi, cioe' quando non devono essere effettuate interruzioni, il programma puo' bloccare molte interruzioni (non tutte!!) per consentire che un particolare processo sia completato.

L' istruzione che consente questo blocco delle interruzioni e':

SEI SEt Interrupt disable flag.

per prevenire quindi le interruzioni.

Stranamente la prima azione che e' necessario fare per ottenere un INTERRUPT e' quella di settare il Flag I (interrupt disable) per mezzo dell' uso dell' istruzione SEI.

Per prevenire Interrupts di troppo, almeno per il momento, dobbiamo effettuare un controllo per vedere se e' il NOSTRO interrupt, cioe' quello che si desidera o si produce con una nostra istruzione, dato che

potrebbero esserci altri interrupts.

Quando ci saremo assicurati che l' interruzione occorsa e' la nostra, cioe' quella che desideravamo, allora possiamo eseguire un CLEAR sul Flag di Interrupt o Flag I (cioe' metterlo a 0) con l' istruzione:

CLI CLear Interrupt disable flag

Cioe' consenti gli interrupt

Se si pensa a quanto detto, capiremo che dobbiamo consentire agli interrupts di essere interrotti.

Non tutti gli interrupts possono essere bloccati settando il Flag I.

Molti casi possono aversi sia durante il lavoro del sistema sia per squilibri o perdite di tensione in rete che richiedono un' azione immediata.

Per consentire al 6510 di distinguere fra questi due tipi di condizioni esistono sull' integrato stesso due PINS uno per ogni tipo di interrupt.

Uno di questi e' il:

NMI o Non Maskable Input pin

che non puo' essere bloccato.

L' altro e' il:

IRQ o Interrupt ReQuest Pin

che puo' essere coperto dal Flag I.

Quando il 6510 riceve un segnale di Interrupt, completa sempre l'istruzione che sta eseguendo prima di fare qualsiasi altra cosa.

Nel caso di un IRQ interrupt dovrebbe, dopo l'ultima istruzione, controllare se e' a 0 il Flag I (CLEAR) e, nel caso non sia a zero continuare fino a quando il programma non esegua il Clear.

Successivamente, prima di entrare nella procedura di Interrupt, il contenuto del PC e' salvato nello Stack, e successivamente viene salvato SR.

Questo salvataggio di registri e' in realta' una mezza misura in quanto, quasi sicuramente, si avranno modifiche ai registri X, Y ed all' Accumulatore.

Successivamente il Flag I e messo a 1 per prevenire altri Interrupts ed allora l'apposito indirizzo della relativa routine di interrupt e' caricato nel PC.

Questo indirizzo viene trovato alle locazioni 65530 e 65531 (\$FFFA e \$FFFB) per NMI e a 65534 e 65535 (\$FFFE e \$FFFF) per l' IRQ.

Queste routines di interrupt devono terminare con:

RTI ReTurn from Interrupt

Trovando questa istruzione il 6510 esegue tre funzioni:

- 1) Ripristina ai valori precedenti l' interrupt il SR.
- 2) Resetta il flag I con un CLI automatico.

- 3) Guarda nello stack per l' indirizzo di ritorno e lo ripristina come in un RTS automatico.

Come abbiamo detto in precedenza il 6510 esegue solo un mezzo lavoro per cui sia A che X e Y devono essere salvati nell' area Stack.

Ricordiamo che X e Y devono essere prima trasferiti sull' Accumulatore, quindi immessi nello stack ( con PHA) e quando si esce dalla routine di interrupt si richiamano i valori per Y e X nell' Accumulatore (con PLA) e quindi si rimetteranno nei rispettivi registri con TAX e TAY.

Il 6510 ha un' altra istruzione di interrupt:

BRK BReak

Quando il 6510 incontra questa istruzione per prima cosa resetta il PC indicizzandolo di una posizione ( in modo tale che il PC punti al byte seguente l' istruzione BRK ), immagazzina questo indirizzo nello Stack.

Setta poi il Flag di Break (B Flag) che e' il bit 4 dell' SR ed immagazzina anche questo nello Stack.

Dopo di cio' il 6510 eseguirà un normale interrupt IRQ usando i vettori IRQ presenti ( come abbiamo visto ) in \$FFFE (LSB) e \$FFFF (MSB).

Successivamente la routine IRQ controllerà da cosa derivi questo interrupt, cioè se e' un vero IRQ o una istruzione BRK.

Di norma se la routine di IRQ scopre che era un' istruzione BRK, si verterà riportati in ambito Basic,

## CORSO DI ASSEMBLER CBM64

avremo un clear di schermo e apparira' il solito Ready. Usando pero' il nostro programma Assembler otteremo un ingresso nel monitor.

Per controllare cio', eseguire il seguente programma:

Programma 62

```
10 LDA #90
20 STA 1024
30 LDA #1
40 STA 55296
50 BRK
```

Dopo aver stampato un quadri bianco in 1024 questo programma saltera' al Monitor.

Usando il nostro programma Assembler per disassemblare i codici di indirizzo \$FFFE, \$FFFF dovrete ormai essere capaci di trovare questi indirizzi e di seguire quindi le operazioni di questa routine.

L' elemento essenziale e' l' istruzione JMP 790 che troverete in 65368 (\$FF58).

Prima di caricare il programma Assembler gli indirizzi 790 e 791 contenevano gli indirizzi della routine che torna al Basic con READY, mentre invece ora contengono gli indirizzi di entrata al monitor.

Quindi questi indirizzi ci sono stati messi dall' Assembler.

## NUMERI RELATIVI

In tutti gli esercizi matematici visti fino a questo momento, sono stati trattati solo numeri positivi.

Perciò ogni processo aritmetico è stato preso in considerazione come trattamento di insiemi di 8 bit.

Tuttavia, nel caso che debba essere usato un intero negativo, uno di questi bit deve essere utilizzato per indicare che stiamo rappresentando un numero negativo.

Il bit più a sinistra nella struttura del Byte, cioè il bit 7, è utilizzato per questo motivo. Viene cioè messo a 0 se il numero che deve essere rappresentato è positivo e a 1 se questo numero è negativo.

Usando questo metodo potremo, rappresentare valori compresi tra +127 e -128.

Infatti considerato che deve essere rappresentato anche lo 0 (ZERO), l'insieme di numeri rappresentati è 256.

Uno dei problemi che si pongono con l'uso di questo metodo è che, in teoria, si possono avere due rappresentazioni per lo 0, cioè -0 e +0 :

$$+0 = 00000000$$

$$-0 = 10000000$$

Per superare questo problema i numeri negativi sono rappresentati nella forma:

### COMPLEMENTO A DUE

che potrà sembrare una forma strana ma che funziona. Vediamo come.

## CORSO DI ASSEMBLER CBM64

Prendiamo il numero 38 (decimale) che in binario e' 00100110.

Per convertirlo nella sua forma negativa complemeto a due e' necessario cambiare tutti gli 0 in 1 e viceversa. Cioe' nel suo complemento a uno:

00100110 diventa 11011001

Successivamente si aggiunge 1:

```
11011001 +
          1
-----
11011010
```

Infatti:

- 38 = 11011010

Per comprendere il significato di questo modo di lavorare vediamo alcuni esempi:

1) 38 - 38 che da 0. Infatti:

```
-38 = 11011010
+38 = 00100110
--  -----
00  00000000
```

## CORSO DI ASSEMBLER CBM64

2) 43 - 38

```
-38 = 11011010
+43 = 00101011
--  -----
5    00000101
```

3) 24 - 38

```
+24 = 00011000
-38 = 11011010
--  -----
-14  11110010
```

Nell' ultima risposta abbiamo un 1 nel bit 7 per cui siamo in presenza di un numero negativo in complemento a due. Per trovarne il valore assoluto, per prima cosa troviamo il complemento a 1:

```
11110010  00001101
```

Aggiungiamo 1:

```
00001101+
          1
-----
00001110= -14
```

## GLI OVERFLOW

Nei Bytes che rappresentano numeri con segno, come abbiamo visto in precedenza, possiamo immagazzinare un valore non superiore a +127. Percio' ogni tentativo di immetterci un numero di dimensioni maggiori provochera' un riporto entro il bit 7 (CARRY) o, come si dice in questo caso un OVERFLOW.

Prendiamo la somma  $100 + 30$

|     |   |          |
|-----|---|----------|
| 100 | = | 01100100 |
| 30  | = | 00011110 |
| --- |   | -----    |
| 130 |   | 10000010 |

Ma per quanto abbiamo visto 10000010 e' un numero negativo, perche' nel bit 7 e' presente un 1. Effettuare quindi il complemento a 1 e poi aggiungere 1.

Il 6510 manipola questo tipo di situazione eseguendo il MONITORING dell' Accumulatore e, quando siamo in presenza di un OVERFLOW, settando il Flag di Overflow o FLAG V.

Questo flag puo' essere controllato dalle seguenti istruzioni:

BVC Branch on oVerflow Clear

BVS Branch on oVerflow Set

## CORSO DI ASSEMBLER CBM64

BVC controlla il contenuto del Flag di overflow e se non e' settato (cioe' V=0) esegue un salto.

BVS le stesse operazioni di cui sopra solo che esegue il salto se V=1.

Quando e' necessario eseguire processi aritmetici in precisione multipla con interi con segno, il bit 7 deve essere trattato come un Carry interno.

Allora se ci troveremo in presenza di un Overflow, questo dovra' essere trasferito nel Byte piu' significativo.

Il seguente programma illustra l' uso di BVC per il controllo di Overflow.

### PROGRAMMA 63

```
10 *=828
20 CLC
30 LDA #1
40 SOMMA ADC #1
50 BVC SOMMA
60 STA 1024
70 LDX #1
80 STA 55296
90 RTS
```

Quando questo programma gira il contenuto dell' Accumulatore e' incrementato progressivamente fino a quando i 7 bits piu' a destra non sono riempiti con 1. Al successivo incremento il settimo bit e' resettato e

viene generato un Carry che entra nel settimo bit. Cio' setta il bit di riporto ed arresta il salto, consentendo al programma di girare fino a RTS.

Allo stesso modo che accade con altri Flags esistono provvedimenti per il controllo del flag di Overflow che puo' essere messo a 0 (CLEAR) con l'istruzione:

CVL CLear the oVerflow flag

Tuttavia, diversamente dagli altri Flag, il flag di Overflow non puo' essere messo a 1 (cioe' settato).

### VISUALIZZAZIONE DEI NUMERI

In tutti gli esempi numerici visti fino a questo momento, le uscite su schermo sono state su codice CBM 64.

Mentre e' possibile interpretare queste visualizzazioni ricorrendo ad una tavola, e' ovviamente necessario in programma visualizzare numeri come numeri in base 10.

La maggiore complicazione che questa procedura comporta sta nel fatto che mentre il codice di visualizzazione del CBM64 e' una effettiva rappresentazione in base 2, per cui per rappresentare numeri fra 0 e 255 e' necessario un solo Byte, quando si desidera visualizzare in base 10 sono necessari tre caratteri ( o Bytes) per rappresentare lo stesso valore.

Il programma seguente serve per eseguire questo tipo di conversione.

Per prima cosa controlla se il numero e' maggiore di 200

## CORSO DI ASSEMBLER CBM64

( se il primo digit e' 2 )) o se e' minore di 200 ma maggiore di 100.

Esegue poi lo stesso controllo per le decine e le unita' ed usa lo stack per immagazzinare il resto del numero mentre aggiunge la costante di conversione (48) all' accumulatore per cambiare il valore binario nell' equivalente valore da visualizzare.

Nel seguente esempio il numero che deve essere visualizzato ( 152 ) e' caricato in accumulatore all' inizio del programma.

### PROGRAMMA 64

|     |           |                                   |
|-----|-----------|-----------------------------------|
| 1   | *=828     |                                   |
| 10  | LDX #1    | Carica il valore 1 in X.          |
| 20  | LDA #152  | Immetti il numero 152 in A        |
| 30  | CMP #200  | Confronta A con 200.              |
| 40  | BCC SALTO | Salta se e' inferiore a 200.      |
| 50  | SBC #200  | Rimuovi il digit piu' a sinistra. |
| 60  | PHA       | Immagazzina il resto nello stack. |
| 70  | LDA #50   | Carica A con "2" per 2 x 100.     |
| 80  | STA 1024  | Visualizza A.                     |
| 90  | STX 55296 | Carica il bianco in RAM.          |
| 100 | PLA       | Ritrova A dallo Stack.            |
| 110 | JMP TENS  | Salta alla routine.               |
| 120 | SALTO CLC | Clear del Carry.                  |
| 130 | CMP #100  | Confronta A con 100.              |
| 140 | BCC TENS  | Salta se inferiore a 100.         |
| 150 | SBC #100  | Rimuovi il digit piu' a sinistra. |
| 160 | PHA       | Immetti il resto nello Stack.     |
| 170 | LDA #49   | Carica A con "1" per 1 x 100.     |
| 180 | STA 1024  | Stampa A sullo schermo.           |
| 190 | STX 55296 | Carica il colore bianco in RAM.   |
| 200 | PLA       | Ritrova A dallo Stack.            |
| 210 | TENS CLC  | Clear del Carry.                  |

## CORSO DI ASSEMBLER CBM64

|               |                                          |
|---------------|------------------------------------------|
| 220 LDY #0    | Fissa Y a 0.                             |
| 230 CMP #9    | Confronta A con 9.                       |
| 240 BCC TENS0 | Salta se A e' minore di 9.               |
| 250 LOOP INY  | Incrementa Y.                            |
| 260 SBC #10   | Sottrai 10 da A.                         |
| 270 CMP #9    | Confronta A con 9.                       |
| 280 BCS LOOP  | Salta se A e' piu' grande di 9.          |
| 290 TENS0 PHA | Carica A nello Stack.                    |
| 300 TYA       | Trasferisci Y in A.                      |
| 310 ADC #48   | Aggiungi la costante di conversione ad A |
| 320 STA 1025  | Visualizza A.                            |
| 330 STX 55297 | Carica il bianco nella RAM colore.       |
| 340 PLA       | Ritrova A dallo stack.                   |
| 350 ADC #48   | Aggiungi la costante di conversione ad A |
| 360 STA 1026  | Visualizza A.                            |
| 370 STX 55298 | Immetti il bianco nella RAM colore.      |
| 380 RTS       |                                          |

Quando questo programma gira sara' visualizzato, come al solito in alto a sinistra dello schermo un 152 in bianco.

In linea generale questo programma puo' essere usato come subroutine di un programma piu' generale per visualizzare il contenuto dell' accumulatore.

# LEZIONE SEDICESIMA

## NUMERI IN VIRGOLA MOBILE

Quando si lavora in Basic le costanti binarie in virgola mobile hanno 10 digits di precisione e sono visualizzate in 9 digits.

I loro esponenti hanno un range da -38 a +37.

Ogni valore e' immagazzinato in 6 bytes consecutivi e, per facilitarne la manipolazione, esistono due "ACCUMULATORI" nelle locazioni di memoria da 97 a 102 (\$61 a \$66) e da 105 a 110 (\$69 a \$6E). Questi sono normalmente conosciuti come:

FAC Floating Point Accumulator

AFAC Alternative Floating Point Accumulator.

Per immagazzinare un numero fino a 10 digits, quando e' visualizzato in base 10, il FAC usa solo sei Bytes.

Quando un programma Basic gira, potrete osservare che numeri molto grandi o molto piccoli sono espressi in forma esponenziale o notazione scientifica.

Percio' 4079.013 e' espresso come 4079013E+ 4 che significa  $4079013 \times 10$  alla quarta.

0.0000417 puo' scriversi come 4.17E- 05 oppure come 4.17 x 10 alla -5.

Questo tipo di rappresentazione contiene due parti.

## CORSO DI ASSEMBLER CBM64

Prendendo il primo caso, la prima parte , .4079013 e' chiamata MANTISSA e la seconda, il +5 di 10 alla quinta, ESPONENTE.

Queste due parti sono immagazzinate in binario nel FAC nella seguente maniera:

A) La MANTISSA BINARIA e' immagazzinata nei quattro Bytes centrali di FAC e AFAC.

Il segno della mantissa e' immagazzinato nel sesto Byte in cui a "1" nel bit 1 corrisponde una mantissa negativa e a "0" nello set dei bit corrisponde una mantissa positiva.

b) L' ESPONENTE BINARIO e' immagazzinato nel primo Byte di FAC e AFAC.

Poiche' e' necessario disporre della possibilita' di immagazzinare sia esponenti negativi che positivi e' necessario eseguire il complemento a 128.

Per questo motivo un esponente di 20 sara' immesso come  $120 + 20 = 148$ , mentre un esponente negativo, come -20, sara'  $128 - 20 = 108$

Quando carica un numero in virgola mobile il Basic normalizza la sua rappresentazione binaria e quindi il suo digit piu' a sinistra e' sempre 1.

Prendiamo per esempio il numero + 1400 (\$0578), espresso in binario e':

0000 0101 0111 1000

In questa forma il numero binario ha un esponente di 2 ed un punto binario implicito ( naturalmente un numero

binario ha un punto binario invece di un punto decimale ed e' conosciuto anche come RADIX ) alla destra del digit meno significativo:

0000 0101 0111 1000

### IL COMANDO USR

Questo comando consente il trasferimento di dati tra il FAC ed un programma in codice macchina.

Per esempio, la linea B=USR(Q) in un programma Basic consente al sistema di immettere il valore di Q entro il FAC.

Questo allora salta alla routine in codice macchina il cui indirizzo e' trovato in 785 (\$0311) come LSB e 786 (\$0312) come MSB.

Si presume che abbiate immesso una routine in codice macchina in memoria che inizi a quell' indirizzo e usi i valori di 785 e 786 per puntare alla routine.

Quando la vostra routine usa FAC utilizzerà il valore che era in Q alla chiamata di USR.

Quando il particolare processo aritmetico e' stato terminato (cioe' ne siamo usciti), la vostra routine dovrebbe lasciare la risposta in FAC e questo sarà immesso in B dal Basic.

Naturalmente si può usare la funzione USR con lo stesso metodo e sistema di altre funzioni.

Per esempio PRINT USR (P:2) visualizzerà il risultato della routine in codice macchina che sarà stata iniziata con il FAC che conteneva il valore immagazzinato in P aumentato di 2.

Per controllare questo processo, può essere messo un

## CORSO DI ASSEMBLER CBM64

numero in FAC per mezzo della funzione USR e dopo visualizzato.

Lo faremo con i due programmi seguenti:

### PROGRAMMA 65

```
20000 PRINT "clear di schermo"
20010 POKE 785,60
20020 POKE 786,3
20030 B=1400
20040 Q=USR(B)
20050 PRINT"Q=";Q
```

### NOTA

L' indirizzo \$033C e' ottenuto con :

```
60=$3C
3=$03
```

### PROGRAMMA 65/B

#### RTS

Al RUN, l' indirizzo \$033C (828) viene caricato nelle locazioni 785 e 786 dalle linee 20010 e 20020.

Quando viene eseguita la linea 20040, l' argomento B (1400) e' caricato nel FAC. Quindi il controllo viene preso dal programma in codice macchina a \$033C.

La routine in codice macchina non puo' a questo punto

## CORSO DI ASSEMBLER CBM64

modificare il valore di FAC, ma eseguendo RTS sara' restituito il controllo al Basic che a sua volta immettera' il contenuto di FAC in Q.

La linea 20050 stampa il valore immagazinato in Q che non e' stato modificato dalla routine in codice macchina.

Questa routine offre un sistema di caricare un numero qualsiasi, che sia valido in Basic, entro il FAC.

Offre anche un sistema per esaminare il contenuto di FAC.

Questo non e' cosi' chiaro come si potrebbe credere, perche' molti comandi Basic usano FAC durante la loro esecuzione, cosi' anche un comando PEEK cambia il suo contenuto.

Tuttavia, se il contenuto e' esaminato in codice macchina, immediatamente dopo essere stato settato, puo' essere visto prima che il Basic ci rimetta le mani.

Per far questo il programma 8.6b dovrebbe essere modificato per esaminare le locazioni da \$61 a \$66 (da 97 a 102) per dopo visualizzarle. Cio' viene fatto nel programma seguente.

### PROGRAMMA 66

```
10 *=828
20 LDX #6
30 PIPPO LDY #1
40 LDA 96,X
50 STA 1024,X
60 TYA
70 STA 55296,X
80 DEX
90 BNE PIPPO
100RTS
```

Stampa del contenuto di FAC sullo schermo.

Come il programma appena visto visualizza il contenuto in codice CBM, lo stesso programma potrebbe essere modificato per decodificare questo codice.

### PROGRAMMA 67

```
20000 PRINT"home"  
20010 POKE 785,60  
20020 POKE786,3  
20030 B=1400  
20040 A=USR(B)  
20050 PRINT"A=";A  
20060 FORX=0T05  
20070 PRINT PEEK (1025 +X);" ";  
20080 NEXT X
```

Questo programma semplicemente guarda (PEEKs) le locazioni che visualizzano il contenuto e stampa le risposte.

### NOTA

Da osservare che il contenuto fra apici della linea 20000 sta ad indicare che deve essere premuto il tasto CLR/HOME, ma senza lo SHIFT.

Quando gira sara' visualizzato:

A = 1400

ed i contenuti:

139 175 0 0 0 47

### SUBROUTINES PER VIRGOLA MOBILE

Vediamo ora cosa ci offre il sistema operativo della Commodore per manovrare con maggiore facilità queste complesse operazioni a sei Bytes.

Per usare queste routines prima di tutto è necessario sapere dove sono, cioè a quale indirizzo trovarle.

A oggi esse hanno avuto quattro indirizzi:

OLD ROM

BASIC 2.0

BASIC 4.0

BASIC V2

le prime due sono delle serie PET/CBM e l'ultima per il VIC-20.

Sono state rilocate ancora una volta per il CBM64.

Gli indirizzi riportati in questo libro sono per i modelli CBM64 attualmente sul mercato.

## CORSO DI ASSEMBLER CBM64

Tuttavia, dato le precedenti esperienze, nulla vieta che in futuro la Commodore, e sempre per il CBM64 decida di cambiare piu' o meno profondamente il Basic e rilocarle da altra parte, magari uguali e con le stesse funzioni, ma con indirizzi diversi.

In appendice e' riportata una tavola, che crediamo utilissima, che riporta l' indirizzo ed il contenuto di queste routines, le applicazioni ,i registri interessati al loro uso, gli errori nei quali eventualmente possiamo incorrere durante l' esecuzione di esse oltre alle routines preparatorie e conseguenti.

Ancora qualcosa prima di addentrarci nella spiegazione di alcune di queste routines.

Si deve anche conoscere da dove queste routines raccolgono i loro dati e dove depositano poi il risultato se si desidera usarle con sicurezza.

Molte di loro iniziano con una piccola sezione di inizializzazione che prepara i dati e li deposita nella giusta posizione per lavoro.

La subroutine che trasferisce i dati dalla memoria nell' AFAC, per esempio, incomincia con il trasferire i suoi indirizzi di data in 31 (\$1F) e 35 (\$23) dall' Accumulatore e dal registro Y. Per cui quando parte da \$BA8C attende di trovare gli indirizzi in A e Y.

A \$BA90 (47760) incomincia la routine vera e propria e allora ricava i suoi indirizzi dati da 34 (\$22) e 35 (\$23).

Allora puo' inserirsi facilmente con gli indirizzi in A e Y, oppure un po' di bytes dopo con i suoi indirizzi in \$22 e \$23.

Un interessante esercizio potrebbe essere quello di disassemblare questa routine, con il comando Disassemble del nostro Assemblatore e con indirizzo di partenza 47756 e fine 47798 , e studiarne il funzionamento

attraverso i vari passi.

Proviamo a scrivere un programma per inserire due numeri in un altro programma in codice macchina. Eseguire quindi una moltiplicazione fra i due numeri ed effettuare la radice quadrata della somma. Visualizzarne le risposte in Basic.

Non e' facile come sembra!!!

#### PROGRAMMA 6S

```

10 *=828
20 JSR $BC0C
30 RTS
40 JSR $BA2B
50 JSR $BF71
60 RTS
70 END
990 REM
1000 REM PROGRAMMAZIONE
1001 REM
1005 PRINT
1010 POKE 785,60
1020 POKE 786,3
1030 INPUT "PRIMO NUMERO";B
1040 A=USR (B)
1050 POKE 785,64
1060 POKE 786,3
1070 INPUT "SECONDO NUMERO";D
1080 C=USR (D)
1090 PRINT "C=";C

```

Dare il comando Assemble e quindi RUN 1000 per far

## CORSO DI ASSEMBLER CBM64

girare il programma Basic.

Vediamo passo passo come funziona questo programma.

```
1005 PRINT
1010 POKE 785,60
1020 POKE 786,3
```

Con questi comandi vengono immessi i valori che stanno ad indicare l' inizio del programma in Linguaggio Macchina della routine richiamata dal comando Basic `USR`.

```
1030 INPUT "PRIMO NUMERO";B
1040 A=USR(B)
```

Viene effettuato l' `INPUT` del primo numero su cui operare e quindi trasferito in `FAC` dal comando `USR`, che a sua volta passa il controllo al programma in `LM` allocato nella zona in cui puntavano le due locazioni 785 e 786. Quindi il computer eseguirà:

```
JSR $BCOC
RTS
```

che trasferisce il contenuto del `FAC` in `AFAC` (`JSR $BCOC`) e quindi ritornerà al Basic con il comando `RTS`. Tornata in ambito Basic il programma verrà ripreso dal punto dell' interruzione cioè dalla riga 1050.

```
1050 POKE 785,64
1060 POKE 786,3
1070 INPUT "SECONDO NUMERO";D
1080 C=USR(D)
```

Le prime due righe di questo pezzo di programma servono

## CORSO DI ASSEMBLER CBM64

come per quelle viste prima per comunicare i parametri al comando USR, in altre parole per dirgli dove si trova la subroutine in LM.

Viene fatto quindi l'immissione del secondo numero e successivamente trasferito tale valore in FAC saltando al seguente codice macchina:

```
JSR $BA2B
JSR $BF71
RTS
```

che provvede a moltiplicare il contenuto di FAC per AFAC (JSR \$BA2B) e ne estrae quindi la radice quadrata (JSR \$BF71) e ritorna, con RTS, a quella parte di programma in Basic che provvede a stampare il contenuto del FAC in conseguenza del comando USR con:

```
1090 PRINT "C=";C
```

Questo era l'intenzione, tuttavia non funziona. Perché?

Non funziona perché la seconda parte del programma ritiene che il contenuto di FAC rimanga fermo, mentre invece cambia continuamente mentre gira il programma Basic.

Dopo la linea 1040 il FAC contiene il valore di B e dopo l'esecuzione di JSR \$BCOC sia FAC che AFAC contengono B. Tuttavia nell'esecuzione delle linee da 1050 a 1080 il S.O. utilizza FAC ed AFAC e di conseguenza ne cambia i contenuti.

Per questo motivo, quando viene chiamata in esecuzione la routine JSR \$BA2B il contenuto di FAC ed AFAC non è quello atteso.

## CORSO DI ASSEMBLER CBM64

Il problema puo' essere superato salvando AFAC in memoria mentre si ritorna al Basic cosa che puo' essere fatta mediante la subroutine di indirizzo \$BBC7.

Questa subroutine copia infatti il contenuto di AFAC nei 5 Bytes di memoria che iniziano all' indirizzo immagazzinato nelle locazioni di memoria \$49 e \$4A.

Il seguente programma ne illustra un esempio immettendo AFAC da \$0384 in poi.

### PROGRAMMA 69

```
10 LDA #132
20 STA $49
30 LDA #3
40 STA $4A
50 JSR $BBC7
60 RTS
```

questa azione puo' essere controllata facendo girare un programma in modo diretto come il seguente:

```
FOR X=0TO5:PRINT PEEK(900+X);:NEXT
```

Per poter utilizzare questa subroutine nel programma e' necessario, dopo aver salvato i contenuti di AFAC in memoria fare in modo che dopo la riga 1080 questi valori si trovino nuovamente in AFAC.

Questo puo' essere fatto usando la Subroutine di indirizzo \$BA8C (decimale 47756).

Perche' questa subroutine possa operare e' necessario che essa sappia dove trovare i dati. Questa informazione viene data caricando l' indirizzo del primo Byte di dati nell' Accumulatore (la parte LSB) e nel Registro Y (la parte MSB).

Un programma di RELOAD AFAC per i dati da 900 in poi potrebbe essere:

```
LDA #132
LDY #3
JSR $BA8C
RTS
```

A questo punto aggiungeremo queste due subroutines al nostro programma e otterremo quindi:

#### PROGRAMMA 70

```
10  *=828
20  JSR $BC0C
30  JSR PIPPO
40  RTS
50  JSR GIGI
60  JSR $BA2B
70  JSR $BF71
80  RTS
90  PIPPO LDA #132
100 STA $49
110 LDA #3
120 STA $4A
130 JSR $BBC7
140 RTS
150 GIGI LDA #132
160 LDY #3
170 JSR $BA8C
180 RTS
190 END
```

## CORSO DI ASSEMBLER CBM64

```
999 REM
1000 REM PROGRAMMA BASIC
1001 REM
1005 PRINT
1010 POKE 785,60
1020 POKE 786,3
1030 INPUT "PRIMO NUMERO";B
1040 A=USR (B)
1050 POKE 785,64
1060 POKE 786,3
1070 INPUT "SECONDO NUMERO";D
1080 C=USR (D)
1090 PRINT "C=";C
```

### NOTA

A questo programma dare per prima cosa il comando Assemble e quindi RUN 1000 per far girare il programma Basic.

### ADDIZIONE

Vediamo, iniziando da questa, altre subroutines. Usando la routine di indirizzo \$B86A (47210) i numeri nel formato in virgola mobile (FLOATING POINT) in FAC ed AFAC sono sommati fra loro ed il risultato della somma caricato in FAC.

Facciamo un esempio con i seguenti programmi:

## PROGRAMMA 71

```
10 *=828
20 LDA #132
30 STA $49
40 LDA #3
50 STA $44
60 JSR $BBC7 !TRASFERISCI FAC IN MEMORIA
70 RTS
80 LDA #132
90 LDY #3
100 JSR $BA8C !RIMETTI FAC IN MEMORIA.
110 JSR $B86A !ROUTINE DI SOMMA
120 RTS
130 END
20000 PRINT "clear/home "
20010 POKE 785,60
20020 POKE 786,3
20030 INPUT B
20040 A=USR (B)
20060 POKE 785,72
20070 POKE 786,3
20080 INPUT D
20090 C=USR(D)
20100 PRINT "C=";C
```

Il programma richiede l' input di due numeri e li restituisce addizionati.

Al solito dare il comando Assemble e poi RUN 20000.

**SOTTRAZIONE**

I programmi precedenti possono essere usati per

## CORSO DI ASSEMBLER CBM64

dimostrare questa operazione inserendo la subroutine di indirizzo \$B853 (47187) alla riga 110

```
110 847 JSR $B853
```

Anche questo dopo il RUN 20000 porra' due richieste di input. Le risposte saranno la sottrazione del secondo valore inserito dal primo.

### DIVISIONE

Usiamo ancora una volta i programmi precedenti per dimostrare l'uso della routine di divisione di indirizzo \$BB12 (47890) rimpiazzando la riga 110 come sopra.

```
110 847 JSR $BB12
```

eseguire RUN 20000.

Il programma eseguirà la divisione fra il primo dato inserito ed il secondo.

### POTENZE

La routine di elevamento a potenza è di indirizzo \$BF7B (49019). Utilizzare i programmi precedenti rimpiazzando ancora una volta la riga 110.

## CORSO DI ASSEMBLER CBM64

110 847 JSR \$BF7B

Dopo il RUN 20000 il primo numero inserito sara' elevato alla potenza del secondo.

Altre routine necessitano di un solo input come:

### LOG

La subroutine e' a \$B9EA (47594) e calcola il logaritmo naturale o in base E. I programmi seguenti ne dimostrano l' uso.

#### PROGRAMMA 72

```
5 *=828
10 JSR $B9EA
20 RTS
30 END
20000 PRINT"clear"
20010 POKE 785,60
20020 POKE 786,3
20030 INPUT B
20040 A=USR(B)
20050 PRINT "A=";A
```

Dopo aver dato il comando Assemble, con un RUN 20000 si attivano entrambe le routines che consentiranno di stampare il logaritmo in base E del numero inserito con l' input di 20030.

### USO DELLA STAMPANTE

Se si desidera stampare i listati dei programmi e' sufficiente utilizzare la procedura standard del Basic che riportiamo in breve di seguito. Per una gestione approfondita di TUTTE le periferiche si consiglia LE PERIFERICHE COMMODORE ed EVM.

OPEN 4,4 (Return) che serve per aprire il canale di comunicazione con la stampante.

CMD 4 (Return) per passare in collegamento diretto.

Dopo di cio' si puo' scegliere tra questa serie dicomandi quale digitare.

### LIST

Per avere semplicemente il listato del programma sia in Basic che in Assembler mnemonico.

### FORMAT

Per ottenere un listato incolonnato come abbiamo visto per questo comando.

### DISASSEMBLE (LOC. INIZIO), (LOC. FINE)

Per ottenere un listato disassemblato di un programma in LM.

### TABLE

Per ottenere i nomi di tutte le Labels e le loro

## CORSO DI ASSEMBLER CBM64

posizioni nella memoria. Questo comando diretto puo' essere utilizzato SOLO dopo che e' stato eseguito il processo di assemblaggio.

Terminato di stampare per chiudere il canale aperto in precedenza con la stampante digitare il comando, sempre diretto:

PRINT# 4 (Return)

# SOLUZIONE ESERCIZI

## «ESERCIZI PROPOSTI»

### NOTA

Naturalmente le presentate sono soluzioni POSSIBILI, ma i relativi problemi potrebbero essere risolti con vari metodi.

Anzi e' bene che si provi a risolverli anche in altra maniera.

### Esercizio 1

Poiche' l' accumulatore contiene un 1 sembra ragionevole usarlo per visualizzare una A bianca.

```
10 *=828
20 LDA #1
30 STA 1024
35 LDA #5
40 STA 55296
50 RTS
```

### Esercizio 2

Poniamo di scrivere il nome FRED in nero. La soluzione potrebbe essere la seguente:

## CORSO DI ASSEMBLER CBM64

```
10  *=828
20  LDA #6
30  STA 1024
40  LDA #0
50  STA 55296
60  LDA #18
70  STA 1025
80  LDA #0
90  STA 55297
100 LDA #5
110 STA 1026
120 LDA #0
130 STA 55298
140 LDA #4
150 STA 1027
160 LDA #0
170 STA 55299
180 RTS
```

La funzione che abbiamo eseguito e' stata quella di separare i colori dall' inserimento dei caratteri.

### Esercizio 3

```
10  *=828
20  LDA #24
30  STA 1024
40  STA 1063
50  STA 1984
60  STA 2023
70  LDA #2
80  STA 55296
90  STA 55335
```

## CORSO DI ASSEMBLER CBM64

```
100 STA 56256
110 STA 56295
120 RTS
```

### Esercizio 5

```
10  *=828
20  LDA #26
30  LDX #1
40  STX 900
50  TAX
60  LDA 900
70  STA 1024
80  STX 2023
90  STA 55296
100 STA 56295
110 RTS
```

### Esercizio 6

```
10  *=828
20  LDA #90
30  LDX #42
40  LDY #5
50  STX 900
60  TAX
70  TYA
80  LDY 900
90  STA 1024
100 STA 1063
110 STX 1984
120 STY 2023
```

```
130 STA 55296
140 STA 55335
150 STA 56256
160 STA 56295
170 RTS
```

### Esercizio 7

A)

```
10 =828
20 LDA #3
30 JSR 900
40 STA 1024
50 LDA #1
60 STA 55296
70 RTS
```

B)

```
10 *=900
20 STA 950
30 ADC 950
40 RTS
```

### Esercizio 8

```
10 *=828
20 LDY #100
30 ROUT DEY
40 BEQ SALTO
```

```
50  JMP ROUT
60  SALTO STY 1024
70  STY 55296
80  RTS
```

## Esercizio 9

```
10  *=828
20  LDA #83
30  STA 890
40  LDY #0
50  INCREM INY
60  CPY 890
70  BEQ SALTO
80  JMP INCREM
90  SALTO STY 1034
100 LDA #4
110 STA 55306
120 RTS
```

## Esercizio 10

```
10  *=828
20  LDX #90
30  DECRX DEX
40  CPX 900
50  BPL DECRX
60  STX 1024
70  STX 55296
80  RTS
```

## Esercizio 11

```
10 *=828
20 LDY #100
30 SALTO LDA #90
40 STA 1023,Y
50 LDA #1
60 STA 55295,Y
70 DEY
80 BNE SALTO
90 RTS
```

## Esercizio 12

```
10 *=828
20 LDX #100
30 STX 900
40 LDX #0
50 ROUT LDA #42
60 STA 1023,X
70 LDA #1
80 STA 55295,X
90 INX
100 CPX 900
110 BNE ROUT
120 RTS
```

## Esercizio 13

```
10 *=828
20 CLC
```

## CORSO DI ASSEMBLER CBM64

```
30 CLD
40 LDA #07
50 ADC #$FA
60 STA 1026
70 LDX #1
80 STX 55298
90 LDA #$18
100 ADC #$2A
110 STA 1024
120 STX 55296
130 RTS
```

### Esercizio 14

```
10 *=528
20 CLD
30 SEC
40 LDA #32
50 SBC #88
60 STA 1036
70 LDX #1
80 STX 55308
90 LDA #3
100 SBC #2
110 STA 1034
120 STX 55306
130 RTS
```

### Esercizio 15

```
10 *=828
20 CLD
```

```
30  CLC
40  LDA #$2C
50  ADC #$90
60  STA 900
70  LDA #$01
80  ADC #$01
90  STA 901
100 SEC
110 LDA 900
120 SBC #$F4
130 STA 1041
140 LDX #1
150 STX 55313
160 LDA 901
170 SBC #$01
180 STA 1040
190 STX 55312
200 RTS
```

## Esercizio 16

|       |       |       |
|-------|-------|-------|
| A = 1 | B = 0 | C = 0 |
| A = 0 | B = 1 | C = 0 |
| A = 1 | B = 1 | C = 1 |

## Esercizio 17

|     |            |
|-----|------------|
| 149 | = 10010101 |
| 52  | = 00110100 |

AND tra 149 e 52 = 00010100 (20)

## CORSO DI ASSEMBLER CBM64

### Esercizio 18

I)

```
10 *=528
20 LDA #100
30 AND #87
40 STA 1024
50 LDX #1
60 STX 55296
70 RTS
```

II)

```
10 *=528
20 LDA #75
30 ORA #27
40 STA 1024
50 LDX #1
60 STX 55296
70 RTS
```

III)

```
10 *=828
20 LDA #99
30 EOR #57
40 STA 1024
50 LDX #1
60 STX 55296
70 RTS
```

IV)

```
10 *=828
20 LDA #100
30 AND #87
40 EOR #94
50 STA 1024
60 LDX #1
70 STX 55296
80 RTS
```

Esercizio 19

```
10 *=828
20 LDA #134 !86 IN BCD
30 AND #15
40 STA 1025
50 LDX #1
60 STX 55297
70 LDY #4
80 LDA #134
90 SALTO LSR A
100 DEY
110 BNE SALTO
120 STA 1024
130 STX 55296
140 RTS
```

Esercizio 20

Scritto per moltiplicare i numeri 3 e 4

```

10  *=828
20  LDX #3
30  STX 901
40  LDX #4
50  STX 902
60  LDY #5
70  LDA #0
80  CLC
90  INCREM LSR 902
100 BCC SALTO
110 CLC
120 ADC 901
130 SALTO ASL 901
140 DEY
150 BNE INCREM
160 STA 1024
170 LDX #1
180 STX 55296
190 RTS

```

### Esercizio 21

```

10  *=828
20  LDX #80
30  LDY #2
40  JMP LOOP3
50  LOOP1 LDA #83
60  STA 1153,X
70  TYA
80  STA 55455,X
90  DEX
100 BNE LOOP1

```

```
110 JMP FINE
120 LOOP2 LDA #90
130 STA 1023,X
140 TYA
150 STA 55295,X
160 DEX
170 BNE LOOP2
172 LDX #120
175 DEY
180 JMP LOOP1
190 LOOP3 LDA #42
200 STA 1423,X
210 TYA
220 STA 55695,X
230 DEX
240 BNE LOOP3
250 LDX #120
260 DEY
270 JMP LOOP2
280 FINE RTS
```

## Esercizio 22

```
10 *=828
20 LDX #32
30 STX 900
40 SALTO JSR 65508 !$FFE4
50 BEQ SALTO
60 JSR 65490 !$FFD2
70 CMP 900
80 BNE SALTO
90 LDA #13
100 JSR 65490 !$FFD2
110 RTS
```

## CORSO DI ASSEMBLER CBM64

### Esercizio 23

```
10  *=828
20  LDX #4
30  SALVAX TXA
40  PHA
50  PHP
60  SALTO ISR $FFE4
70  BEQ SALTO
80  JSR $FFD2
90  PLP
100 PLA
110 TAX
120 DEX
130 BNE SALVAX
140 RTS
```

### NOTA FINALE

Come abbiamo piu' volte ripetuto per un impiego approfondito di questo nuovo linguaggio e quindi dei metodi di programmazione che lo accompagnano non si puo' prescindere da una conoscenza dell' intero Sistema Operativo del CBM64.

Relativamente al computer la EVM ha pubblicato il gia' citato manuale IL SISTEMA OPERATIVO DEL CBM64 che contiene:

-L' intero listato del S.O. stesso

-Il listato, sempre disassemblato, dell' interprete BASIC.

-Il CROSS REFERENCE per poter osservare accuratamente l'interazione delle operazioni.

Al termine del manuale viene riportata una completa mappa di memoria a confronto fra i sistemi operativi dei computer Commodore e precisamente: CBM64 - VIC20 - PET - CBM 4.0 in modo tale che sia possibile riportare anche i programmi da un computer all' altro.

Altro manuale per chi volesse approfondire il collegamento con la principale periferica, il disco e', sempre a cura dell' EVM "I SEGRETI DEL 1541".

Questo manuale riporta:

-Il Sistema Operativo del disco completamente disassemblato e commentato.

-Le tecniche di programmazione avanzata su disco.

-Una serie di applicazioni e di esempi come ad esempio il DISK MONITOR

Il manuale viene completato con lo schema elettrico dell' unita' a dischi.

APPENDICI

E

TAVOLE

# CORSO DI ASSEMBLER CBM64

## LE ROUTINES KERNAL

| NOME   | INDIRIZZO |       | FUNZIONE                                 |
|--------|-----------|-------|------------------------------------------|
|        | HEX       | DEC   |                                          |
| ACPTR  | \$FFA5    | 65445 | INGRESSO BYTE DALLA PORTA SERIALE        |
| CHKIN  | \$FFC6    | 65478 | APRE UN CANALE PER INPUT                 |
| CHKOUT | \$FFC9    | 65481 | APRE UN CANALE PER OUTPUT                |
| CHRIN  | \$FFCF    | 65487 | INGRESSO DI UN CARATTERE DA UN CANALE    |
| CHROUT | \$FFD2    | 65490 | OUTPUT DI UN CARATTERE A UN CANALE       |
| CIOUT  | \$FFA8    | 65448 | OUTPUT DI UN BYTE ALLA PORTA SERIALE     |
| CINT   | \$FF81    | 65409 | INIZIALIZZA L' EDITOR DI SCHERMO         |
| CLALL  | \$FFE7    | 65511 | CHIUDI TUTTI I CANALI E I FILES          |
| CLOSE  | \$FFC3    | 65475 | CHIUDI UN DATO FILE LOGICO               |
| CLRCHN | \$FFCC    | 65484 | CHIUDI I CANALI IN I/O                   |
| GETIN  | \$FFE4    | 65508 | RICEVE UN CARATT. DALLA CODA DI TASTIERA |
| IOBASE | \$FFF3    | 65523 | RIPORTA INDIRIZZO BASE PERIF. I/O        |
| IOINIT | \$FF84    | 65412 | INIZIALIZZA INPUT/OUTPUT                 |
| LISTEN | \$FFB1    | 65457 | COMANDO AL BUS SERIALE PER LISTEN        |
| LOAD   | \$FFD5    | 65493 | CARICA RAM DA PERIFERICA                 |
| MEMBOT | \$FF9C    | 65436 | LEGGE/FISSA IL MINIMO DI MEMORIA         |
| MEMTOP | \$FF99    | 65433 | LEGGE/FISSA IL MASSIMO DI MEMORIA        |
| OPEN   | \$FFC0    | 65472 | APRE UN FILE LOGICO                      |
| PLOT   | \$FFF0    | 65520 | LEGGE/FISSA POSIZIONE DEL CURSORE        |
| RAMTAS | \$FF87    | 65415 | INIZ. RAM, DISPONE PER BUFFER NASTRO     |
| RDTIM  | \$FFDE    | 65502 | LEGGE OROLOGIO IN TEMPO REALE            |
| READST | \$FFB7    | 65463 | LEGGE IN I/O LO STATO DELLA PAROLA       |

# CORSO DI ASSEMBLER CBM64

| NOME   | INDIRIZZO |       | FUNZIONE                                  |
|--------|-----------|-------|-------------------------------------------|
|        | HEX       | DEC   |                                           |
| RESTOR | \$FF5A    | 65418 | REINTEGRA ASSENZA DEI VETTORI I/O         |
| SAVE   | \$FFD5    | 65496 | SALVA RAM SU PERIFERICA                   |
| SCNKEY | \$FF9F    | 65439 | SCANSIONE DI TASTIERA                     |
| SCREEN | \$FFED    | 65517 | RIPORTA ORGANIZZ. SCHERMO DI X,Y          |
| SECOND | \$FF93    | 65427 | INVIA L'INDIRIZZO SECON. DOPO<br>LISTEN   |
| SETLFS | \$FFBA    | 65466 | FISSA INDIR LOGICO E SECONDARIO           |
| SETMSG | \$FF90    | 65420 | CONTROLLO MESSAGGI KERNAL                 |
| SETNAM | \$FFBD    | 65469 | FISSA IL NOME DEL FILE                    |
| SETTIM | \$FFDB    | 65499 | FISSA L' OROLOGIO (R.T.CLOCK)             |
| SETTMO | \$FFA2    | 65442 | FISSA IL TIMEOUT (fuori tempo)SUL<br>BUS. |
| STOP   | \$FFE1    | 65505 | SCANSIONE DEL TASTO DI STOP               |
| TALK   | \$FFB4    | 65460 | COM. SUL BUS SER. PER TALK                |
| TKSA   | \$FF96    | 65430 | INVIA IND. SEC. DOPO TALK                 |
| UDTIM  | \$FFEA    | 65514 | INCREMENTA OROLOGIO                       |
| UNLSN  | \$FFAE    | 65454 | COMANDO AL BUS SERIALE PER<br>UNLISTEN    |
| UNTLK  | \$FFAB    | 65451 | COMANDO AL BUS SERIALE PER UNTALK         |
| VECTOR | \$FF8D    | 65421 | LEGGE/FISSA GLI I/O VETTORIZZATI          |

DESCRIZIONE DELLE ROUTINES KERNAL

B-1 Nome della funzione: ACPTR

FUNZIONE:riceve dati dal bus seriale

INDIRIZZO:\$FFA5 - 65445

REGISTRI DI COMUNICAZIONE:..A

ROUTINES DI PREPARAZIONE: TALK, TKSA

ERRORI:Vedi READST

BYTES DI STACK USATI: 13

REGISTRI USATI: A,X

DESCRIZIONE :Questa e' la routine che si usa quando si desidera ricevere informazioni da una periferica attraverso il BUS seriale, per esempio da disco.

Questa routine riceve un Byte di dati dal Bus usando un HANDSHAKING pieno ed il dato e' riportato in Accumulatore.

La routine TALK deve essere chiamata in funzione prima di ordinare alla periferica di inviare dati sul Bus.

Se la periferica in ingresso necessita di un comando secondario, questo deve essere inviato usando la routine TKSA prima di ACPTR.

Se ci sono errori saranno riportati nella PAROLA di STATO (STATUS WORD) il cui contenuto potra' essere letto

dalla routine READST.

#### MODO DI UTILIZZO

- 1) Ordinare alla periferica collegata al Bus seriale di prepararsi ad inviare dati al CBM64 usando le routines TALK e TKSA.
- 2) Saltare, con JSR, quindi alla routine ACPTR.
- 3) Immagazzinare o utilizzare i dati ricevuti.

B-2 Nome della funzione: CHKIN

FUNZIONE: Apre un canale per INPUT

INDIRIZZO: \$FFC6 - 65478

REGISTRI DI COMUNICAZIONE: .X

ROUTINES DI PREPARAZIONE: (OPEN)

ERRORI: Vedi nota

BYTES DI STACK USATI: Nessuno

REGISTRI USATI: .A, .X

DESCRIZIONE: Un qualsiasi File logico che sia stato aperto per mezzo della routine OPEN puo' essere definito come un canale di Input per mezzo di questa routine.

## CORSO DI ASSEMBLER CBM64

Naturalmente la periferica sul canale deve essere una periferica in input, perche' altrimenti avremo un errore e la CHKIN non avra' effetto.

Se si stanno ricevendo dati da una qualsiasi altra parte che non sia la tastiera, questa routine (OPEN) deve essere chiamata prima di usare sia le routine CHRIN che GETIN per l' Input dei dati.

Se si desidera usare l' Input da tastiera, e nessun altro canale di input e' aperto. allora la chiamata a questa Routine e alla routine OPEN non e' necessaria.

Quando questa routine e' utilizzata con una periferica sul bus Seriale, essa invia automaticamente l' indirizzo di chiamata ( e l' indirizzo secondario se questo e' specificato in OPEN) sul Bus.

### MODO DI UTILIZZO

- 1) Eseguire l'OPEN del file logico (se necessario)
- 2) Caricare il registro X con il numero del file logico che deve essere usato.
- 3) Saltare a questa routine.

### POSSIBILI ERRORI

- #3: File non aperto
- #5: Periferica non presente ( o non collegata)
- #6: Il file non e' un file input.

B-3 Nome della funzione: CHKOUT

## CORSO DI ASSEMBLER CBM64

FUNZIONE: Apre un canale per OUTPUT

INDIRIZZO: \$FFC9 - 65481

REGISTRI DI COMUNICAZIONE: X

ROUTINES DI PREPARAZIONE: OPEN

ERRORI: 0,3,5,7

BYTES DI STACK USATI: 4+

REGISTRI USATI: A,X

DESCRIZIONE: Un qualsiasi numero di File logico che sia stato creato dalla routine OPEN puo' essere definito come un canale di OUTPUT.

Percio' la periferica deve essere una periferica in OUTPUT cioe' in uscita perche' in caso contrario avremo una segnalazione di errore.

Questa routine (CHKOUT) deve essere messa in funzione prima che un qualsiasi dato sia inviato ad una periferica (naturalmente in uscita) a meno che non si desideri usare lo schermo in funzione di periferica in uscita.

Quando e' usata per aprire un canale per una periferica sul Bus seriale, questa nostra routine inviera' automaticamente l' indirizzo di LISTEN specificato dalla routine OPEN e l' indirizzo secondario se esiste.

### MODO DI UTILIZZO

- 1) Utilizzare la routine OPEN per specificare un numero

## CORSO DI ASSEMBLER CBM64

di file logico, l' indirizzo di LISTEN e l' indirizzo secondario se necessario.

2) Caricare il registro X con il numero di File logico usato nel comando OPEN.

3) Chiamare ora la CHKOUT

### POSSIBILI ERRORI

#3: File non aperto

#5: Periferica non presente ( o non collegata)

#7: Non e' un file in uscita

B-4 Nome della funzione: CHRIN

FUNZIONE: Riceve un carattere da un canale di Input

INDIRIZZO: \$FFCF - 65487

REGISTRI DI COMUNICAZIONE: A

ROUTINES DI PREPARAZIONE: (OPEN, CHKIN)

ERRORI: Vedi READST

BYTES DI STACK USATI: 7+

REGISTRI USATI: A, X

DESCRIZIONE: Questa routine riceve un byte di dati da un canale gia' selezionato per mezzo della routine CHKIN

## CORSO DI ASSEMBLER CBM64

come canale in INPUT.

Se CHKIN non e' stata usata per definire un diverso canale di input allora i dati saranno attesi da tastiera.

Il Byte di dati e' caricato in accumulatore ed il canale resta aperto.

L' ingresso da tastiera e' manipolato in maniera particolare.

Per prima cosa e' attivato il cursore che lampeggera' fino alla digitazione di un ritorno carrello da tastiera (cioe' fino a quando non sia premuto il RETURN).

Tutti i caratteri della linea ( max 88) sono immagazzinati nel BASIC INPUT BUFFER.

Questi caratteri sono recuperati ad uno ad uno per mezzo di tanti salti a questa routine quanti sono questi caratteri.

Quando viene incontrato il ritorno carrello l' intera linea e' stata manipolata.

### MODO DI UTILIZZO

Da tastiera

- 1) Rintraccia un Byte di dati per mezzo del salto a questa routine.
- 2) Immagazzina il Byte
- 3) Controlla se e' l' ultimo Byte, cioe' se e' un RETURN
- 4) Se non e' un RETURN, ritorna al passo 1).

## CORSO DI ASSEMBLER CBM64

Da altre periferiche

- 1) Usare OPEN e CHKIN
- 2) Saltare a CHRIN
- 3) Immagazzinare i dati

B-5 Nome della funzione: CHROUT

FUNZIONE: Uscita di un carattere

INDIRIZZO: \$FFD2 - 65490

REGISTRI DI COMUNICAZIONE: A

ROUTINES DI PREPARAZIONE: (CHKOUT, OPEN)

ERRORI: Vedi READST

BYTES DI STACK USATI: 8+

REGISTRI USATI: A

DESCRIZIONE: Questa routine fa uscire un carattere su un canale già aperto.

E' necessario usare le routines OPEN e CHKOUT per fissare un canale di uscita prima di chiamare questa routine.

Nel caso che queste chiamate siano omesse i dati saranno inviati alla periferica base in uscita, cioè la numero 3 il video.

## CORSO DI ASSEMBLER CBM64

I Byte che devono uscire , cioè' che sono in Output sono caricati nell' Accumulatore, viene chiamata la routine CHROUT e successivamente i dati sono inviati alla periferica selezionata, mentre il canale viene lasciato aperto.

### MODO DI UTILIZZO

- 1) Usare la CHROUT se necessario. Vedi quanto già' scritto.
- 2) Caricare i dati nell' accumulatore
- 3) Saltare ora a CHROUT

B-6 Nome della funzione: CIOUT

FUNZIONE: Trasmette un Byte sul bus seriale

INDIRIZZO:\$FFA8 - 65448

REGISTRI DI COMUNICAZIONE: A

ROUTINES DI PREPARAZIONE: LISTEN (SECOND)

ERRORI:Vedi READST

BYTES DI STACK USATI:5

REGISTRI USATI:

## CORSO DI ASSEMBLER CBM64

**DESCRIZIONE:** Questa routine e' utilizzata per inviare informazioni a periferiche collegate al bus seriale. Percio' la messa in funzione di questa routine avra' come conseguenza l' immissione di un byte di dati sul bus seriale usando un HANDSHAKING seriale pieno. Prima di chiamare questa routine, deve essere chiamata la routine LISTEN che ordina' alla periferica sul BUS seriale di tenersi pronta a ricevere i dati. (Se alla periferica necessita un indirizzo secondario questo deve essere inviato attraverso l' utilizzo della routine SECOND che vedremo in seguito). La periferica deve essere in ascolto o sara' generato, attraverso la parola di stato , un errore di fuori tempo (TIMEOUT).

### MODO DI UTILIZZO

- 1) Usare per prima cosa la routine LISTEN e poi la SECOND se necessaria.
- 2) Caricare l' Accumulatore con un Byte di dati.
- 3) Saltare a questa routine per inviare il Byte di dati

B-7 Nome della funzione: CINT

**FUNZIONE:** Inizializza l' editor di schermo e l' integrato 6567

**INDIRIZZO:** \$FF81 - 65409

**REGISTRI DI COMUNICAZIONE:**

## CORSO DI ASSEMBLER CBM64

ROUTINES DI PREPARAZIONE:

ERRORI:

BYTES DI STACK USATI:4

REGISTRI USATI:A, X, Y

DESCRIZIONE: Questa routine abilita l' integrato 6567 (VIDEO CONTROLLER) nel CBM64 per le normali operazioni. Viene inizializzato anche il KERNAL SCREEN EDITOR. Dovrebbe essere chiamata in funzione da un catridge.

MODO DI UTILIZZO

1) Salto a questa routine da cui si dovrebbe passare al RUN automatico

B-8 Nome della funzione: CLALL

FUNZIONE: Chiude tutti i Files

INDIRIZZO:\$FFE7 - 65511

REGISTRI DI COMUNICAZIONE:

ROUTINES DI PREPARAZIONE:

ERRORI:

## CORSO DI ASSEMBLER CBM64

BYTES DI STACK USATI:11

REGISTRI USATI: A, X

DESCRIZIONE: Questa routine serve per chiudere tutti i files aperti.

Quando entra in funzione questa routine i puntatori della tavola dei file aperti sono resettati, chiudendo cosi' tutti i files.

Anche la routine CLRCHN viene chiamata per resettare tutti i canali di I/O.

### MODO DI UTILIZZO

1) Saltare semplicemente a questa routine.

B-9 Nome della funzione: CLOSE

FUNZIONE: Chiude un file logico

INDIRIZZO:\$FFC3 - 65475

REGISTRI DI COMUNICAZIONE: A

ROUTINES DI PREPARAZIONE:

ERRORI:Vedi READST

BYTES DI STACK USATI:2+

REGISTRI USATI:A, X, Y

## CORSO DI ASSEMBLER CBM64

DESCRIZIONE: Questa routine e' utilizzata per chiudere un file logico dopo che tutte le operazioni di I/O sullo stesso file sono state eseguite.

La routine e' chiamata dopo che l' accumulatore e' stato caricato con il numero di file logico che deve essere chiuso.

Naturalmnete questo sara' lo stesso numero usato quando il file era stato aperto con OPEN.

### MODO DI UTILIZZO

1) Caricare l' accumulatore con il numero di file logico che deve essere chiuso.

2) Eseguire la routine.

B-10 Nome della funzione: CLRCHN

FUNZIONE: Pulisci i canali di I/O

INDIRIZZO:\$FFCC - 65484

REGISTRI DI COMUNICAZIONE:

ROUTINES DI PREPARAZIONE:

ERRORI:

BYTES DI STACK USATI:9

REGISTRI USATI: A, X

DESCRIZIONE: Questa routine e' utilizzata per eseguire il CLEAR di tutti i canali aperti e ripristinare gli stessi canali ai loro valori originari.

La periferica normale di INPUT e' 0 (cioe' la tastiera), mentre la periferica normale di OUTPUT e' 3 (cioe' il video).

Se uno dei canali che deve essere chiuso e' su una porta seriale, viene inviato per prima cosa un segnale di UNTALK per eseguire la pulizia del canale di Input o un segnale di UNLISTEN per la pulizia del canale di Output. Non eseguendo la chiamata a questa routine e quindi lasciando gli ascoltatori ( LISTENERS ) attivi sul bus seriale, diverse periferiche possono ricevere gli stessi dati dal CBM64 allo stesso tempo.

Un sistema per utilizzare questa particolarita' potrebbe essere quello di mettere la stampante in TALK e il disco in LISTEN per consentire la stampa diretta di un file disco.

La routine CLRCHN entra automaticamente in funzione dopo l' esecuzione di CLALL.

#### MODO DI UTILIZZO

1) Saltare a questa routine con JSR

B-11 Nome della funzione: GETIN

FUNZIONE: Riceve un carattere da periferica.

## CORSO DI ASSEMBLER CBM64

INDIRIZZO:\$FFE4 - 65508

REGISTRI DI COMUNICAZIONE: A

ROUTINES DI PREPARAZIONE: CHKIN OPEN

ERRORI:Vedi READST

BYTES DI STACK USATI:5+

REGISTRI USATI: A, (X, Y)

DESCRIZIONE: Se il carattere e' da tastiera, questa routine prende un carattere dalla coda di tastiera (KEYBOARD QUEUE) e lo riporta nell' Accumulatore come valore ASCII.

Se il buffer ( cioe' la coda di tastiera che e' appunto un buffer) e' vuota allora il carattere caricato nell' Accumulatore sara' zero.

I caratteri sono immessi nella coda di tastiera utilizzando sia una parte HARDWARE (INTERRUPT DRIVEN KEYBOARD) sia la routine di scnsione della tastiera SCNKEY.

Il buffer puo' contenere al massimo 10 caratteri per cui se e' pieno gli altri carateri che si tentera' di immettere saranno ignorati fino a quando almeno un carattere non sia rimosso dalla coda.

Se il canale invece di essere la tastiera e' l' RS-232 allora viene usato solo il registro A e viene riportato un solo carattere e sara' necessario utilizzare READST per il controllo di validita'.

Se il canale invece e' seriale, cassetta o schermo e' chiamata la routine BASIN.

## CORSO DI ASSEMBLER CBM64

### MODO DI UTILIZZO

- 1) Chiamare la routine con una istruzione JSR.
- 2) Eseguire il controllo per 0 nell' Accumulatore
- 3) Manipolare i dati.

B-12 Nome della funzione: IOBASE

FUNZIONE: Definisce la pagina di memria I/O

INDIRIZZO:\$FFF3 - 65523

REGISTRI DI COMUNICAZIONE: X, Y

ROUTINES DI PREPARAZIONE:

ERRORI:

BYTES DI STACK USATI:2

REGISTRI USATI:X, Y

DESCRIZIONE: Questa routine fissa i registri X e Y all' indirizzo della sezione di memoria che definisce dove sono localizzate le periferiche I/O.

Questo indirizzo puo' essere utilizzato come linea di deviazione (OFFSET) per accedere alla memoria disegnata per le periferiche I/O del CBM64.

La linea di deviazione e' il numero di locazioni dall' inizio della pagina sulla quale si desidera che i

## CORSO DI ASSEMBLER CBM64

registri I/O siano immessi.

Naturalmente il registro X conterra' il Byte di indirizzo piu' basso mentre Y la parte alta dello stesso indirizzo.

### MOD0 DI UTILIZZO

- 1) Eseguire il JSR a questa routine
- 2) Immagazzinare i due registri X e Y in due indirizzi consecutivi.
- 3) Caricare il registro Y con OFFSET
- 4) Accedera a quella locazione di I/O.

## CORSO DI ASSEMBLER CBM64

B-13 Nome della funzione: IOINIT

FUNZIONE: Inizializza periferiche I/O

INDIRIZZO:\$FF84 - 65412

REGISTRI DI COMUNICAZIONE:

ROUTINES DI PREPARAZIONE:

ERRORI:

BYTES DI STACK USATI:

REGISTRI USATI: A, Y, X

DESCRIZIONE: Questa routine inizializza tutte le periferiche di I/O e le routines.

E' normalmente chiamata come parte di una procedura di inizializzazione di un programma su cartridge

B-14 Nome della funzione: LISTEN

FUNZIONE: Invia un comando di ascolto ad una periferica sul Bus seriale

INDIRIZZO:\$FFB1 - 65457

REGISTRI DI COMUNICAZIONE:A

ROUTINES DI PREPARAZIONE:

## CORSO DI ASSEMBLER CBM64

ERRORI: Vedi READST

BYTES DI STACK USATI:

REGISTRI USATI: A

DESCRIZIONE: Questa routine ordina ad una periferica sul Bus seriale di ricevere dati.

L' Accumulatore deve essere caricato con un numero compreso fra 0 e 31 prima di chiamare questa routine.

LISTEN eseguirà un OR logico sul numero bit per bit per convertirlo in un indirizzo di ascolto e poi trasmetterà questo dato come comando sul bus seriale.

La periferica specificata si metterà allora in modo di ascolto e sarà pronta per ricevere informazioni.

### MODO DI UTILIZZO

1) Carica l' Accumulatore con il numero della periferica alla quale deve essere inviato l' ordine di ascolto.

2) Vai alla routine con un comando di JSR.

B-15 Nome della funzione: LOAD

FUNZIONE: Carica RAM da una periferica

INDIRIZZO: \$FFD5 - 65493

REGISTRI DI COMUNICAZIONE: A, X, Y

## CORSO DI ASSEMBLER CBM64

ROUTINES DI PREPARAZIONE: SETLFS, SETNAM

ERRORI: Vedi READST

BYTES DI STACK USATI:

REGISTRI USATI: A, Y, X

DESCRIZIONE: Questa routine carica Bytes di dati da una qualsiasi periferica in INPUT direttamente entro la memoria del CBM64.

Puo' anche essere usata per una operazione di verifica che avviene confrontando i dati presenti sulla periferica con quelli in memoria e lasciando i dati in memoria inalterati.

L' Accumulatore deve essere messo a 0 per un' operazione di LOAD o messo a 1 per un' operazione di verifica.

Se la periferica in Input e' aperta con un' indirizzo secondario di 0, allora sara' ignorata la testata (HEADER) dell' informazione.

In questo caso i registri X e Y devono contenere l' indirizzo di partenza per LOAD.

Se la periferica e' collegata con un ' indirizzo secondario 1 0 2, allora i dati saranno caricati in memoria con partenza dall' indirizzo specificato dalla testata.

Questa routine inoltre riporta l' indirizzo della piu' alta locazione di RAM caricata.

Prima di chiamare questa routine e' necessario chiamare le routines SETLFS e SETNAM.

### NOTA

Non si puo' eseguire il LOAD da Tastiera (0), RS-232 (2)

o schermo (3).

#### MODO DI UTILIZZO

- 1) Eseguire per prima cosa le routines SETLFS e SETNAM
- 2) Mettere A a 0 per LOAD o a 1 per VERIFY.
- 3) Saltare alla routine con JSR

B-16 Nome della funzione: MEMBOT

FUNZIONE: Fissa la parte piu' bassa della memoria

INDIRIZZO:\$FF9C - 65436

REGISTRI DI COMUNICAZIONE: X Y

ROUTINES DI PREPARAZIONE:

ERRORI:

BYTES DI STACK USATI:

REGISTRI USATI: Y, X

DESCRIZIONE: Questa routine e' usata per fissare la parte piu' bassa della memoria.

Se il bit di Carry dell' Accumulatore e' a 1 quando viene chiamata questa routine, allora un puntatore che indica il Byte piu' basso della RAM e' riportato in X e

## CORSO DI ASSEMBLER CBM64

Y.

Normalmente sul CBM 64 il puntatore e' \$0800 cioe' 2048. Se il bit di Carry dell' Accumulatorre e' a 0 quando viene chiamata questa routine, allora il valore dei registri X e Y sara' trasferito rispettivamente al piu' basso e piu' alto Byte del puntatore all' inizio della RAM.

### MODO DI UTILIZZO

Per leggere il punto di partenza della RAM:

- 1) Mettere il Carry a 1
- 2) Chiamare la Routine

Per leggere il punto di partenza della Memoria:

- 1) Mettere il Carry a 0
- 2) Chiamare la routine

B-17 Nome della funzione: MEMTOP

FUNZIONE: Fissa la parte alta della memoria

INDIRIZZO: \$FF99 - 65433

REGISTRI DI COMUNICAZIONE: X Y

## CORSO DI ASSEMBLER CBM64

ROUTINES DI PREPARAZIONE:

ERRORI:

BYTES DI STACK USATI: 2

REGISTRI USATI: Y, X

DESCRIZIONE: Questa routine e' utilizzata per fissare il punto massimo della memoria RAM.

Il funzionamento e' simile alla routine precedente (MEMBOT).

Infatti anche in questo caso quando si utilizza questa Routine con il bit di Carry dell' Accumulatore a 1, il puntatore alla fine della memoria RAM e' caricato nei registri X e Y.

Quando invece la si utilizza con il bit di carry a 0, allora il contenuto dei registri X e Y e' caricato nel puntatore al massimo della memoria.

B-18 Nome della funzione: OPEN

FUNZIONE: Apre un file logico

INDIRIZZO:\$FFC0 - 65472

REGISTRI DI COMUNICAZIONE:

ROUTINES DI PREPARAZIONE: SETLFS , SETNAM

ERRORI: Vedi READST

## CORSO DI ASSEMBLER CBM64

BYTES DI STACK USATI:

REGISTRI USATI: A, Y, X

DESCRIZIONE: Questa routine e' utilizzata per eseguire la funzione di apertura di un File logico.

Non appena il file logico e' stato fissato questi puo' essere utilizzato per operazioni di I/O.

Molte delle Routines del sistema operativo fanno uso di OPEN.

Non sono necessari argomenti o operandi ma prima di utilizzare questa routine sara' necessario metterne in funzione altre due cioe' la SETLFS e la SETNAM.

### MODO DI UTILIZZO

- 1) Utilizzare la SETLFS
- 2) Utilizzare quindi la SETNAM
- 3) Saltare a OPEN

B-13 Nome della funzione: PLOT

FUNZIONE: Legge e fissa la posizione del cursore

INDIRIZZO: \$FFFF0 - 65520

REGISTRI DI COMUNICAZIONE: A, X, Y

ROUTINES DI PREPARAZIONE:

## CORSO DI ASSEMBLER CBM64

ERRORI:

BYTES DI STACK USATI: 2

REGISTRI USATI: A, Y, X

DESCRIZIONE: Quando si salta a questa routine con il carry dell' accumulatore a 1, allora la posizione attuale del cursore , nelle sue coordinate X e Y sara' caricata nel registri Y e X dove Y sara' il numero di colonna del cursore ( da 0 a 79) e X il numero di riga occupato dal cursore ( da 0 a 24).

Se invece il carry e a 0 allora verranno letti i valori dei registri X e Y e il cursore posizionato a quei valori.

### MODO DI UTILIZZO

Lettura della locazione del cursore:

1) Metti a 1 il carry

2) Vai alla routine

3) Ricava la posizione del cursore dalla lettura del contenuto dei registri X e Y per le posizioni X e Y cioe' ascissa e ordinata.

Fissa l' indirizzo del cursore

1) Metti a 0 il carry

## CORSO DI ASSEMBLER CBM64

- 2) Carica nei registri X e Y la posizione del cursore
- 3) Esegui la routine

B-20 Nome della funzione: RAMTAS

FUNZIONE: Controlla le RAM, fissa aree per buffer nastro e schermo

INDIRIZZO: \$FF87 - 65415

REGISTRI DI COMUNICAZIONE: A, X, Y

ROUTINES DI PREPARAZIONE:

ERRORI:

BYTES DI STACK USATI: 2

REGISTRI USATI: A, X, Y

DESCRIZIONE: Questa routine e' utilizzata per controllare la memoria RAM e fissare i puntatori della memoria sia in alto che in basso.

Esegue anche il clear delle locazioni \$0000 fino a \$0101 e da \$0200 a \$03FF.

Fissa anche il buffer di cassetta e fissa la locazione base di schermo a partire da \$0400.

Normalmente questa routine e' chiamata come parte di un processo di inizializzazione di un Cartridge.

## CORSO DI ASSEMBLER CBM64

B-21 Nome della funzione: RDTIM

FUNZIONE: Legge l'orologio in tempo reale

INDIRIZZO: \$FFDE - 65502

REGISTRI DI COMUNICAZIONE: A, X, Y

ROUTINES DI PREPARAZIONE:

ERRORI:

BYTES DI STACK USATI: 2

REGISTRI USATI: A, X, Y

DESCRIZIONE: Questa routine e' utilizzata per leggere il clock o orologio di sistema.

La risoluzione del clock, cioe' il tempo minimo e' di 1/60mo di secondo.

Il risultato della lettura di questa routine e' di 3 Bytes che sono riportati rispettivamente nell' Accumulatore, nel registro X e Y..

Operando con questi tre registri e, come vedremo poi con la routine SETTIM e' possibile leggere e variare il contenuto dell' orologio del sistema.

B-22 Nome della funzione: READST

FUNZIONE: Legge lo STATUS WORD

## CORSO DI ASSEMBLER CBM64

INDIRIZZO: \$FFB7 - 65463

REGISTRI DI COMUNICAZIONE: A

ROUTINES DI PREPARAZIONE:

ERRORI:

BYTES DI STACK USATI: 2

REGISTRI USATI: A

DESCRIZIONE: Questa routine riporta lo stato attuale delle periferiche in I/O nell' accumulatore.

E' utilizzata di norma dopo ogni colloquio con le periferiche e riporta le informazioni sullo stato delle periferiche stesse o eventuali errori incontrati durante operazioni di I/O.

I BITS di questa routine, caricati nell' Accumulatore contengono le informazioni illustrate nella seguente tavola:

| POSIZ.<br>BIT | VALORE<br>NUM. | DESCRIZIONE              |
|---------------|----------------|--------------------------|
| 0             | 1              | Time out in scrittura    |
| 1             | 2              | " " lettura              |
| 2             | 4              | Blocco corto             |
| 3             | 8              | Blocco lungo             |
| 4             | 16             | Err. di lettura irrecup. |

## CORSO DI ASSEMBLER CBM64

| POSIZ.<br>BIT | VALORE<br>NUM. | DESCRIZIONE        |
|---------------|----------------|--------------------|
| 5             | 32             | Errore di CHECKSUM |
| 6             | 64             | END OF FILE        |
| 7             | -128           | END OF TAPE *      |

\* Questo tipo di errore sul Bus seriale sta a significare invece un : DEVICE NOT PRESENT.

### MODO DI UTILIZZO

- 1) Richiamare questa routine
- 2) Decodificare l' informazione presente nell' Accumulatore ricavandone i significati appena esposti per utilizzarli eventualmente in programma.

B-23 Nome della funzione: RESTOR

FUNZIONE: Reintegra i vettori di sistema.

INDIRIZZO: \$FF8A - 65418

REGISTRI DI COMUNICAZIONE:

ROUTINES DI PREPARAZIONE:

ERRORI:

BYTES DI STACK USATI: 2

## CORSO DI ASSEMBLER CBM64

REGISTRI USATI: A, X, Y

DESCRIZIONE: Questa routine reintegra i valori mancanti dei vettori di tutto il sistema usati sia nelle KERNAL che nel BASIC come routines e come interrupts.

MODO DI UTILIZZO

1) Saltare a questa routine

B-24 Nome della funzione: SAVE

FUNZIONE: Salva la memoria RAM su periferica

INDIRIZZO: \$FFD8 - 65496

REGISTRI DI COMUNICAZIONE: A, X, Y

ROUTINES DI PREPARAZIONE: SETLFS, SETNAM

ERRORI: Vedi READST

BYTES DI STACK USATI:

REGISTRI USATI: A, X, Y

DESCRIZIONE: Questa routine e' utilizzata per eseguire l' operazione di SAVE di una parte di memoria. La memoria e' salvata da un' indirizzo indiretto in pagina 0 specificato dall' Accumulatore a un' indirizzo

## CORSO DI ASSEMBLER CBM64

immagazzinato nei registri X e Y.

Sara' quindi inviato ad un File logico su una periferica.

Le routines SETLFS e SETNAM devono essere utilizzate prima di accedere a questa routine.

Tuttavia non e' necessario dare un nome al file che si desidera salvare su cassetta, mentre e' necessario per qualsiasi altra periferica.

### MODO DI UTILIZZO

1) Usare SETLFS e SETNAM

2) Caricare due locazioni consecutive di memoria in pagina 0 con un puntatore all' inizia della memoria che si desidera salvare. Ricordarsi che il formato deve essere quello standard del 6502, cioe' prima il Byte basso e poi il Byte alto dell' indirizzo.

3) Caricare l' Accumulatore con il singolo Byte di pagina zero deviato al puntatore.

4) Caricare i registri X e Y rispettivamente con il Byte basso, Byte alto dell' indirizzo di fine memoria. (cioe' della fine della memoria che si vuole caricare).

5) Saltare a SAVE

B-25 Nome della funzione: SCNKEY

FUNZIONE: Esegue la scansione di tastiera

## CORSO DI ASSEMBLER CBM64

INDIRIZZO: \$FF9F - 65439

REGISTRI DI COMUNICAZIONE:

ROUTINES DI PREPARAZIONE: IOINIT

ERRORI:

BYTES DI STACK USATI: 5

REGISTRI USATI: A, X, Y

DESCRIZIONE: Questa routine esegue la scansione (cioe' la lettura) della tastiera e controlla se ci sono tasti premuti.

E' la stessa routine chiamata per mezzo della manipolazione di Interrupt.

Se un tasto e' premuto, allora il suo valore ASCII e' immesso nella coda di tastiera.

### MODO DI UTILIZZO

1) Salto alla routine stessa

B-26 Nome della funzione: SCREEN

FUNZIONE: Riporta il formato dello schermo

INDIRIZZO: \$FFED - 65517

REGISTRI DI COMUNICAZIONE: X, Y

## CORSO DI ASSEMBLER CBM64

ROUTINES DI PREPARAZIONE:

ERRORI:

BYTES DI STACK USATI: 2

REGISTRI USATI: X, Y

DESCRIZIONE: Questa routine restituisce il formato dello schermo di 40 colonne per 25 linee nei registri X e Y. E' utilizzata o puo' essere utilizzata per vedere che tipo di programma sta girando data la grande flessibilita' del CBM64 ad assumere anche altri sistemi operativi.

MODO DI UTILIZZO

1) Chiamata della routine stessa

## CORSO DI ASSEMBLER CBM64

B-27 Nome della funzione: SECOND

FUNZIONE: Invia un indirizzo secondario per la funzione di ascolto ( LISTEN)

INDIRIZZO: \$FF93 - 65427

REGISTRI DI COMUNICAZIONE: A

ROUTINES DI PREPARAZIONE: LISTEN

ERRORI: Vedi READST

BYTES DI STACK USATI: 8

REGISTRI USATI: A

DESCRIZIONE: Questa routine e' utilizzata per inviare un indirizzo secondario ad una periferica in I/O dopo che e' stata effettuata una chiamata alla routine LISTEN e quindi e' stato ordinato alla periferica di porsi in ascolto.

Questa routine non puo' essere usata per inviare un indirizzo secondario dopo un salto alla routine TALK (chiamata).

Normalmente un indirizzo secondario e' usato per comunicare il tipo di informazione che si desidera inviare alla periferica.

## CORSO DI ASSEMBLER CBM64

### MODO DI UTILIZZO

1) Caricare l' Accumulatore con l' indirizzo secondario che deve essere inviato.

2) Eseguire quindi la routine

B-28 Nome della funzione: SETLFS

FUNZIONE: Fissa il file logico ( in maniera completa)

INDIRIZZO: \$FFBA - 65466

REGISTRI DI COMUNICAZIONE: A, X, Y

ROUTINES DI PREPARAZIONE:

ERRORI:

BYTES DI STACK USATI: 2

REGISTRI USATI:

DESCRIZIONE: Questa routine fissa il numero di file logico, l' indirizzo della periferica e l' indirizzo secondario per le altre routines.

Il numero di file logico e' usato dal sistema come chiave di riferimento alla tavola creata dalla routine OPEN file.

L' indirizzo della periferica puo' essere un numero dell' intervallo da 0 a 31.

I seguenti codici sono usati dal CBM64 per gli indirizzi

di tabella:

| INDIRIZZO | PERIFERICA                |
|-----------|---------------------------|
| 0         | TASTIERA                  |
| 1         | DATASETTE                 |
| 2         | PERIFERICA RS232C         |
| 3         | SCHERMO                   |
| 4         | STAMPANTE SUL BUS SERIALE |
| 8         | DISCO CBM SUL BUS SERIALE |

I numeri di periferica uguali o maggiori di 4 si riferiscono a periferiche sul bus seriale.

Un comando alla periferica e' inviato come un indirizzo secondario al bus seriale dopo che il numero della stessa periferica e' stato inviato durante la sequenza di HANDSHAKING ATTENTION.

Se non viene inviato nessun indirizzo secondario, allora il registro Y dovrebbe essere messo a 255.

#### MODO DI UTILIZZO

- 1) Carica l' accumulatore con il numero di file logico.
- 2) Carica il registro X con il numero di device
- 3) Carica il registro Y con il comando.

## CORSO DI ASSEMBLER CBM64

B-29 Nome della funzione: SETMSG

FUNZIONE: Controllo dei messaggi di sistema in uscita

INDIRIZZO: \$FF90 - 65420

REGISTRI DI COMUNICAZIONE: A

ROUTINES DI PREPARAZIONE:

ERRORI:

BYTES DI STACK USATI: 2

REGISTRI USATI: A

DESCRIZIONE: Questa routine controlla la stampa di errore ed i messaggi di controllo delle Kernal routines. Sia la stampa dei messaggi di errore come la stampa dei messaggi di controllo possono essere selezionate, cioè scelte, fissando l' accumulatore quando viene chiamata la routine.

FILE NOT FOUND e' un esempio di messaggio d' errore.

PRESS PLAY ON TAPE e' un esempio di messaggio di controllo.

I bits 6 e 7 di questo valore determinato da dove viene il messaggio.

Se il bit 7 e' a 1, allora verra' stampato un messaggio di errore proveniente dalle KERNAL.

Se il bit 6 e' a 1, viene stampato un messaggio di controllo.

## CORSO DI ASSEMBLER CBM64

### MODO DI UTILIZZO

- 1) Fissare l' accumulatore al valore desiderato.
- 2) Chiamare in funzione la routine

B-30 Nome della funzione: SETNAM

FUNZIONE: Fissa il nome del file

INDIRIZZO: \$FFBD - 65496

REGISTRI DI COMUNICAZIONE: A, X, Y

ROUTINES DI PREPARAZIONE:

ERRORI:

BYTES DI STACK USATI:

REGISTRI USATI:

DESCRIZIONE: Questa routine e' utilizzata per fissare il nome del file per le routine di OPEN, SAVE e LOAD.

L' Accumulatore deve essere caricato con la lunghezza del nome del file.

I registri X e Y devono essere caricati con l' indirizzo del nome del file secondo il formato 6510 cioe' prima il byte basso e poi il byte alto.

L' indirizzo puo' essere un qualsiasi proponibile

## CORSO DI ASSEMBLER CBM64

indirizzo di memoria del sistema dove sia appunto immagazzinata una stringa di caratteri che e' il nome del file.

Se non si desidera nessun nome, allora l' Accumulatore deve essere messo a 0 che rappresentera' un file di lunghezza zero.

In questo caso i registri X e Y possono essere fissati ad un qualsiasi indirizzo di memoria.

### MODO DI UTILIZZO

1) Carica l' Accumulatore con la lunghezza del nome del file.

2) Carica il registro indice X con la parte bassa dell' indirizzo del nome del file

2) Carica il registro indice Y con la parte alta dell' indirizzo del nome del file

3) Esegui SETNAM

B-31 Nome della funzione: SETTIM

FUNZIONE: Fissa i valori del clock di sistema

INDIRIZZO: \$FFDB - 65499

REGISTRI DI COMUNICAZIONE: A, X, Y

ROUTINES DI PREPARAZIONE:

ERRORI:

BYTES DI STACK USATI: 2

REGISTRI USATI:

DESCRIZIONE: L' orologio di sistema e' mantenuto da una routine di interrupt che lo aggiorna ogni sessantesimo di secondo (un "JIFFY" o ciclo ).

Il sistema di clock occupa 3 Bytes che da una capacita' di contare fino a 5.184.000 cicli ( o JIFFY) per un totale di 24 ore dopo di che l' orologio torna a zero.

Prima di chiamare questa routine che serve per modificare i valori dell ' orologio l' accumulatore deve contenere il Byte piu' significativo del tempo fissato nell' orologio, il registro X il secondo Byte ed il registro Y il Byte meno significativo.

MODO DI UTILIZZO

- 1) Carica l' Accumulatore con MSB del numero di 3 byte.
- 2) Carica il regsitro X con il Byte successivo.
- 3) Carica il registro Y con LSB.
- 4) Esegui ora la routine.

B-32 Nome della funzione: SETTMO

FUNZIONE: Fissa il flag di fuori tempo ( TIME-OUT) sul

## CORSO DI ASSEMBLER CBM64

bus IEEE

INDIRIZZO: \$FFA2 - 65442

REGISTRI DI COMUNICAZIONE: A

ROUTINES DI PREPARAZIONE:

ERRORI:

BYTES DI STACK USATI: 2

REGISTRI USATI:

DESCRIZIONE: Questa routine fissa il flag di Fuori tempo per la IEEE.

Quando questo flag e' messo a 1 il CBM64 attendera' una risposta da una periferica sulla IEEE per 64 millisecondi.

Se la periferica non rispondera' al segnale DAV (cioe' DATA ADDRESS VALID) entro questo tempo allora il CBM64 riconoscerà una condizione di errore ed abbandonerà la sequenza di HANDSHAKE.

Quando questa routine e' chiamata ed il bit 7 dell' accumulatore contiene uno 0 allora il TIMEOUT e' abilitato, mentre un 1 nello stesso bit dell' accumulatore lo disabilita.

### NOTA

Questa routine e' usata solo con il cartridge della IEEE.

Il CBM64 usa la possibilita' del TIMEOUT per riconoscere che un file su disco, sempre collegato alla IEEE, non e'

stato trovato

#### MODO DI UTILIZZO

Per fissare il flag di TIMEOUT:

- 1) Mettere a 0 il bit 7 dell' accumulatore.
- 2) Eseguire la routine

Per resettare il flag di TIMEOUT

- 1) Mettere a 1 il bit 7 dell' accumulatore.
- 2) Eseguire la routine

B-33 Nome della funzione: STOP

FUNZIONE: Controlla se il tasto di STOP e' premuto.

INDIRIZZO:\$FFE1 - 65505

REGISTRI DI COMUNICAZIONE: A

ROUTINES DI PREPARAZIONE:

ERRORI:

BYTES DI STACK USATI:

REGISTRI USATI: A, X

## CORSO DI ASSEMBLER CBM64

DESCRIZIONE: Se il tasto di STOP era premuto durante la chiamata alla routine UDTIM, la chiamata a questa routine mette a 1 il flag Z.

Per di piu' i canali saranno resettati per mancanza di valori, mentre tutti gli altri flags rimarranno immutati.

Se il tasto di STOP non era premuto allora l' Accumulatore conterra' un Byte che rappresenta l' ultima riga della scansione di tastiera.

L' utente con questo metodo puo' anche controllare alcuni altri tasti.

### MODO DI UTILIZZO

1) Prima di questa routine dovrebbe essere chiamata la routine UDTIM

2) Chiamare ora STOP

3) Controllare il flag Z.

B-34 Nome della funzione: TALK

FUNZIONE: Comando ad una periferica sul BUS seriale di TALK.

INDIRIZZO:\$FFB4 - 65460

REGISTRI DI COMUNICAZIONE: A

ROUTINES DI PREPARAZIONE:

## CORSO DI ASSEMBLER CBM64

ERRORI: Vedi READST

BYTES DI STACK USATI: 8

REGISTRI USATI: A

DESCRIZIONE: Per utilizzare questa routine per prima cosa l' Accumulatore deve essere caricato con un numero di periferica fra 0 e 31.

Quando e' chiamata questa routine, allora viene eseguito un OR logico bit per bit per convertire il numero della periferica in un indirizzo di chiamata.

Quindi questi dati saranno trasmessi come comando sul Bus seriale.

### MODO DI UTILIZZO

- 1) Carica l' accumulatore con il numero di periferica.
- 2) Salta alla routine.

B-35 Nome della funzione: TKSA

FUNZIONE: Invia un indirizzo secondario ad una periferica dopo la routine TALK

INDIRIZZO: \$FF96 - 65430

REGISTRI DI COMUNICAZIONE: A

ROUTINES DI PREPARAZIONE: TALK

## CORSO DI ASSEMBLER CBM64

ERRORI: Vedi READST

BYTES DI STACK USATI: 8

REGISTRI USATI: A

DESCRIZIONE: Questa routine trasmette un indirizzo secondario ad una periferica in attesa di TALK sul bus seriale.

Prima di chiamarla deve esserci un numero fra 0 e 31 nell' accumulatore.

La routine invia questo numero come un comando di indirizzo secondario sul bus seriale.

TKSA puo' essere messa in funzione dopo la chiamata a TALK mentre non operera' dopo una routine o un comando di LISTEN.

### MODO DI UTILIZZO

- 1) Utilizzare per prima TALK
- 2) Caricare l' accumulatore con l' indirizzo secondario.
- 3) Eseguire TKSA

B-36 Nome della funzione: UDTIM

FUNZIONE: Incrementa l' orologio del sistema

INDIRIZZO: \$FFEA - 65514

## CORSO DI ASSEMBLER CBM64

REGISTRI DI COMUNICAZIONE: A

ROUTINES DI PREPARAZIONE:

ERRORI:

BYTES DI STACK USATI: 2

REGISTRI USATI: A, X

DESCRIZIONE: Questa routine incrementa l' orologio del sistema.

Normalmente questa routine e' chiamata dalla normale routine KERNAL di interrupt ogni sessantesimo di secondo.

Se l' utente si programma da se gli interrupt questa routine DEVE essere chiamata per incrementare il temporizzatore.

Per di piu', se il tasto STOP e' funzionante, cioe' non e' stato disabilitato, deve essere chiamata anche la routine di STOP che abbiamo visto prima.

### MODO DI UTILIZZO

Eseguire la routine alle condizioni dette.

B-37 Nome della funzione: UNLSN

FUNZIONE: Invia un comando di UNLISTEN

## CORSO DI ASSEMBLER CBM64

INDIRIZZO:\$FFAE - 65454

REGISTRI DI COMUNICAZIONE:

ROUTINES DI PREPARAZIONE:

ERRORI:Vedi READST

BYTES DI STACK USATI:8

REGISTRI USATI: A

DESCRIZIONE: Questa routine ordina a, tutti le periferiche sul bus seriale di fermare la ricezione dei dati dal CBM64.

In altre parole chiamando questa routine viene inviato un comando di UNLISTEN sul bus seriale.Cio' naturalmente avra' effetto solo sulle periferiche alle quali era stato in precedenza inviato un comando di LISTEN.

Di norma questa routine e' usata dopo che il CBM64 ha terminato di inviare dati alle periferiche esterne.

Ricordiamo che inviando un comando di UNLISTEN il bus seriale viene reso disponibile per altri usi dal momento che le periferiche aperte sono escluse.

### MODO DI UTILIZZO

Eseguire il salto alla routine.

B-38 Nome della funzione: UNTLK

## CORSO DI ASSEMBLER CBM64

FUNZIONE: Invia un comando di UNTALK

INDIRIZZO:\$FFAB - 65451

REGISTRI DI COMUNICAZIONE:

ROUTINES DI PREPARAZIONE:

ERRORI: Vedi READST  
BYTES DI STACK USATI: 8

REGISTRI USATI: A

DESCRIZIONE: Come la precedente solo che invia un messaggio di UNTALK.

Naturalmente anche in questo caso avremo una disabilitazione delle periferiche dal bus seriale .

### MODO DI UTILIZZO

Salto alla routine

B-39 Nome della funzione: VECTOR

FUNZIONE: Manipola i vettori su RAM

INDIRIZZO:\$FF8D - 65421

REGISTRI DI COMUNICAZIONE: X, Y

ROUTINES DI PREPARAZIONE:

## CORSO DI ASSEMBLER CBM64

ERRORI:

BYTES DI STACK USATI:2

REGISTRI USATI: A, X, Y

DESCRIZIONE: Questa routine manipola tutti i sistemi di indirizzi di salto vettorizzati immagazzinati in RAM.

Chiamando questa routine con il bit di carry dell' accumulatore a 1, l' attuale contenuto dei vettori della RAM viene immagazzinato in una lista a cui puntano X e Y.

Chiamando invece questa routine con lo stesso bit a 0, una lista dell' utente indirizzata dal contenuto dei registri X e Y e' trasferita nel sistema dei vettori RAM

### NOTA

Questa routine richiede cautela nell' uso.

Il miglior sistema e' quello di immettere il contenuto dei vettori in un' area di memoria preventivamente scelta, cambiare quello che si desidera e solo allora copiare il contenuto nell' area RAM.

### MODO DI UTILIZZO

Leggere i vettori RAM

1) Mettere a 1 il carry

2) Fissare i registri X e Y

3) Eseguire la routine

Caricamento dei vettori

1) Mettere a 0 il bit di carry

2) Fissare i registri Y e X

3) Eseguire la routine.

### MESSAGGI DI ERRORE

La seguente e' una lista dei messaggi di errore che possono capitare quando si eseguono delle routine Kernal.

Ricordiamo che se si incappa in un errore durante l'esecuzione di una di queste routine il bit di carry dell' Accumulatore viene messo a 1 ed il numero del messaggio di errore viene immesso nell' accumulatore. Pero' alcuni routines non usano questi messaggi o codici d' errore ma quelli, gia' visti della routine READST.

| NUMERO | SIGNIFICATO                            |
|--------|----------------------------------------|
| 0      | Routine finita con tasto STOP          |
| 1      | Troppi files aperti contemporaneamente |
| 2      | File gia' aperto                       |
| 3      | File non aperto                        |

## CORSO DI ASSEMBLER CBM64

| NUMERO | SIGNIFICATO                                             |
|--------|---------------------------------------------------------|
| 4      | File non trovato                                        |
| 5      | Periferica non presente                                 |
| 6      | Il File non e' in input                                 |
| 7      | Il file non e' in output                                |
| 8      | Manca il nome del file                                  |
| 9      | Numero della periferica illogico                        |
| 240    | Punto massimo della memoria. Cambiare il buffer RS-232. |

## TAVOLE DELLE ISTRUZIONI DEL 6510

## ADC

Addiziona con Carry

| Mnem.        | COD. OPER |     | N.B. | N.CY. | MODI INDIRIZZAMENTO |
|--------------|-----------|-----|------|-------|---------------------|
|              | DEC       | HEX |      |       |                     |
| ADC # OPER   | 105       | 69  | 2    | 2     | Immediato           |
| ADC OPER     | 101       | 65  | 2    | 3     | Pagina Zero         |
| ADC OPER,X   | 117       | 75  | 2    | 4     | Pagina Zero,X       |
| ADC OPER     | 109       | 6D  | 3    | 4     | Assoluto            |
| ADC OPER,X   | 125       | 7D  | 3    | 4*    | Assoluto,X          |
| ADC OPER,Y   | 121       | 79  | 3    | 4*    | Assoluto,Y          |
| ADC (OPER,X) | 97        | 61  | 2    | 6     | (Indiretto, X)      |
| ADC (OPER),Y | 113       | 71  | 2    | 5*    | (Indiretto),Y       |

\*Aggiungere 1 ciclo se salta pagina

## AND

And logico fra accumulatore e memoria

| Mnem.        | COD. OPER |     | N.B. | N.CY. | MODI INDIRIZZAMENTO |
|--------------|-----------|-----|------|-------|---------------------|
|              | DEC       | HEX |      |       |                     |
| AND # OPER   | 41        | 29  | 2    | 2     | Immediato           |
| AND OPER     | 37        | 25  | 2    | 3     | Pagina Zero         |
| AND OPER,X   | 53        | 35  | 2    | 4     | Pagina Zero,X       |
| AND OPER     | 45        | 2D  | 3    | 4     | Assoluto            |
| AND OPER,X   | 61        | 3D  | 3    | 4*    | Assoluto,X          |
| AND OPER,Y   | 57        | 39  | 3    | 4*    | Assoluto,Y          |
| AND (OPER,X) | 33        | 21  | 2    | 6     | (Indiretto, X)      |
| AND (OPER),Y | 49        | 31  | 2    | 5     | (Indiretto),Y       |

\*Aggiungere 1 ciclo se salta pagina

# CORSO DI ASSEMBLER CBM64

## ASL

Esegue uno spostamento verso sinistra

| Mnem.      | COD. OPER |     | N.B. | N.CY. | MODI INDIRIZZAMENTO |
|------------|-----------|-----|------|-------|---------------------|
|            | DEC       | HEX |      |       |                     |
| ASL A      | 10        | 0A  | 1    | 2     | Accumulatore        |
| ASL OPER   | 6         | 06  | 2    | 5     | Pagina Zero         |
| ASL OPER,X | 22        | 16  | 2    | 6     | Pagina Zero,X       |
| ASL OPER   | 14        | 0E  | 3    | 6     | Assoluto            |
| ASL OPER,X | 30        | 1E  | 3    | 7     | Assoluto,X          |

## BCC

Esegue il salto specificato se il Carry=0

| Mnem.    | COD. OPER |     | N.B. | N.CY. | MODI INDIRIZZAMENTO |
|----------|-----------|-----|------|-------|---------------------|
|          | DEC       | HEX |      |       |                     |
| BCC OPER | 144       | 90  | 2    | 2*    | Relativo            |

\*Aggiungere 1 ciclo se il salto e' nella stessa pagina

\*Aggiungere 2 cicli se il salto e' a pagina diversa

## BCS

Esegue il salto specificato se il Carry=1

| Mnem.    | COD. OPER |     | N.B. | N.CY. | MODI INDIRIZZAMENTO |
|----------|-----------|-----|------|-------|---------------------|
|          | DEC       | HEX |      |       |                     |
| BCS OPER | 176       | B0  | 2    | 2*    | Relativo            |

\*Aggiungere 1 ciclo se il salto e' nella stessa pagina

\*Aggiungere 2 cicli se il salto e' a pagina diversa

## CORSO DI ASSEMBLER CBM64

### BEQ

| Esegue il salto se il Flag Z=1 |           |     |      |       |                     |
|--------------------------------|-----------|-----|------|-------|---------------------|
| Mnem.                          | COD. OPER |     | N.B. | N.CY. | MODI INDIRIZZAMENTO |
|                                | DEC       | HEX |      |       |                     |
| BEQ    OPER                    | 240       | F0  | 2    | 2*    | Relativo            |

\*Aggiungere 1 ciclo se il salto e' nella stessa pagina

\*Aggiungere 2 cicli se il salto e' a pagina diversa

### BIT

| Esegue un AND fra A e memoria |           |     |      |       |                     |
|-------------------------------|-----------|-----|------|-------|---------------------|
| Mnem.                         | COD. OPER |     | N.B. | N.CY. | MODI INDIRIZZAMENTO |
|                               | DEC       | HEX |      |       |                     |
| BIT    OPER                   | 36        | 24  | 2    | 3     | Pagina Zero         |
| BIT    OPER                   | 44        | 2C  | 3    | 4     | Assoluto            |

### BMI

| Esegue un salto se il Flag N=1 |           |     |      |       |                     |
|--------------------------------|-----------|-----|------|-------|---------------------|
| Mnem.                          | COD. OPER |     | N.B. | N.CY. | MODI INDIRIZZAMENTO |
|                                | DEC       | HEX |      |       |                     |
| BMI    OPER                    | 48        | 30  | 2    | 2*    | Relativo            |

\*Aggiungere 1 ciclo se il salto e' nella stessa pagina

\*Aggiungere 2 cicli se il salto e' a pagina diversa

## CORSO DI ASSEMBLER CBM64

BNE

| Esegue un salto se il Flag Z=0 |           |     |      |       |                     |
|--------------------------------|-----------|-----|------|-------|---------------------|
| Mnem.                          | COD. OPER |     | N.B. | N.CY. | MODI INDIRIZZAMENTO |
|                                | DEC       | HEX |      |       |                     |
| BNE    OPER                    | 208       | D0  | 2    | 2*    | Relativo            |

\*Aggiungere 1 ciclo se il salto e' nella stessa pagina

\*Aggiungere 2 cicli se il salto e' a pagina diversa

BPL

| Esegue il salto se il Flag N=0 |           |     |      |       |                     |
|--------------------------------|-----------|-----|------|-------|---------------------|
| Mnem.                          | COD. OPER |     | N.B. | N.CY. | MODI INDIRIZZAMENTO |
|                                | DEC       | HEX |      |       |                     |
| BPL    OPER                    | 16        | 10  | 2    | 2*    | Relativo            |

\*Aggiungere 1 ciclo se il salto e' nella stessa pagina

\*Aggiungere 2 cicli se il salto e' a pagina diversa

BRK

| Forza una NMI |           |     |      |       |                     |
|---------------|-----------|-----|------|-------|---------------------|
| Mnem.         | COD. OPER |     | N.B. | N.CY. | MODI INDIRIZZAMENTO |
|               | DEC       | HEX |      |       |                     |
| BRK           | 0         | 00  | 1    | 7     | Implicito           |

## CORSO DI ASSEMBLER CBM64

### BVC

| Esegue il salto se il Flag 0=0 |                      |      |       |                     |  |
|--------------------------------|----------------------|------|-------|---------------------|--|
| Mnem.                          | COD. OPER<br>DEC HEX | N.B. | N.CY. | MODI INDIRIZZAMENTO |  |
| BVC OPER                       | 80 50                | 2    | 2*    | Relativo            |  |

\*Aggiungere 1 ciclo se il salto e' nella stessa pagina

\*Aggiungere 2 cicli se il salto e' a pagina diversa

### BVS

| Salta se il Flag 0=1 |                      |      |       |                     |  |
|----------------------|----------------------|------|-------|---------------------|--|
| Mnem.                | COD. OPER<br>DEC HEX | N.B. | N.CY. | MODI INDIRIZZAMENTO |  |
| BVS OPER             | 112 70               | 2    | 2*    | Relativo            |  |

\*Aggiungere 1 ciclo se il salto e' nella stessa pagina

\*Aggiungere 2 cicli se il salto e' a pagina diversa

### CLC

| Mette a 0 il bit di Carry |                      |      |       |                     |  |
|---------------------------|----------------------|------|-------|---------------------|--|
| Mnem.                     | COD. OPER<br>DEC HEX | N.B. | N.CY. | MODI INDIRIZZAMENTO |  |
| CLC                       | 24 18                | 1    | 2     | Implicito           |  |

## CORSO DI ASSEMBLER CBM64

### CLD

Mette a 0 il bit del Flag D

| Mnem. | COD. OPER<br>DEC HEX | N.B. | N.CY. | MODI INDIRIZZAMENTO |
|-------|----------------------|------|-------|---------------------|
| CLD   | 216 DS               | 1    | 2     | Implicito           |

### CLI

Mette a 0 il bit del Flag I

| Mnem. | COD. OPER<br>DEC HEX | N.B. | N.CY. | MODI INDIRIZZAMENTO |
|-------|----------------------|------|-------|---------------------|
| CLI   | 88 58                | 1    | 2     | Implicito           |

### CLV

Mette a 1 il bit del Flag 0

| Mnem. | COD. OPER<br>DEC HEX | N.B. | N.CY. | MODI INDIRIZZAMENTO |
|-------|----------------------|------|-------|---------------------|
| CLV   | 184 B8               | 1    | 2     | Implicito           |

## CORSO DI ASSEMBLER CBM64

### CMP

| Confronta l' Accumulatore con la Memoria |                      |      |       |                     |  |
|------------------------------------------|----------------------|------|-------|---------------------|--|
| Mnem.                                    | COD. OPER<br>DEC HEX | N.B. | N.CY. | MODI INDIRIZZAMENTO |  |
| CMP # OPER                               | 201 C9               | 2    | 2     | Immediato           |  |
| CMP OPER                                 | 197 C5               | 2    | 3     | Pagina Zero         |  |
| CMP OPER,X                               | 213 D5               | 2    | 4     | Pagina Zero,X       |  |
| CMP OPER                                 | 205 CD               | 3    | 4     | Assoluto            |  |
| CMP OPER,X                               | 221 DD               | 3    | 4*    | Assoluto,X          |  |
| CMP OPER,Y                               | 217 D9               | 3    | 4*    | Assoluto,Y          |  |
| CMP (OPER,X)                             | 193 C1               | 2    | 6     | (Indiretto, X)      |  |
| CMP (OPER),Y                             | 209 D1               | 2    | 5*    | (Indiretto),Y       |  |

\*Aggiungere 1 ciclo se salta pagina

### CPX

| Confronta la Memoria con X |                      |      |       |                     |  |
|----------------------------|----------------------|------|-------|---------------------|--|
| Mnem.                      | COD. OPER<br>DEC HEX | N.B. | N.CY. | MODI INDIRIZZAMENTO |  |
| CPX # OPER                 | 224 E0               | 2    | 2     | Immediato           |  |
| CPX OPER                   | 228 E4               | 2    | 3     | Pagina Zero         |  |
| CPY OPER                   | 236 EC               | 3    | 4     | Assoluto            |  |

## CORSO DI ASSEMBLER CBM64

### CPY

| Confronta la Memoria con Y |                      |    |      |       |                     |
|----------------------------|----------------------|----|------|-------|---------------------|
| Mnem.                      | COD. OPER<br>DEC HEX |    | N.B. | N.CY. | MODI INDIRIZZAMENTO |
| CPY # OPER                 | 192                  | C0 | 2    | 2     | Immediato           |
| CPY OPER                   | 196                  | C4 | 2    | 3     | Pagina Zero         |
| CPY OPER                   | 204                  | CC | 3    | 4     | Assoluto            |

### DEC

| Decrementa di 1 la Memoria |                      |    |      |       |                     |
|----------------------------|----------------------|----|------|-------|---------------------|
| Mnem.                      | COD. OPER<br>DEC HEX |    | N.B. | N.CY. | MODI INDIRIZZAMENTO |
| DEC OPER                   | 198                  | C6 | 2    | 5     | Immediato           |
| DEC OPER,X                 | 214                  | D6 | 2    | 6     | Pagina Zero,X       |
| DEC OPER                   | 206                  | CE | 3    | 6     | Assoluto            |
| DEC OPER,X                 | 222                  | DE | 3    | 7     | Assoluto,X          |

### DEX

| Decrementa X di 1 |                      |    |      |       |                     |
|-------------------|----------------------|----|------|-------|---------------------|
| Mnem.             | COD. OPER<br>DEC HEX |    | N.B. | N.CY. | MODI INDIRIZZAMENTO |
| DEX               | 202                  | CA | 1    | 2     | Implicito           |

## CORSO DI ASSEMBLER CBM64

### DEY

| Decrementa Y di 1 |                      |      |       |                     |
|-------------------|----------------------|------|-------|---------------------|
| Mnem.             | COD. OPER<br>DEC HEX | N.B. | N.CY. | MODI INDIRIZZAMENTO |
| DEY               | 136 88               | 1    | 2     | Implicito           |

### EOR

| Esegue un EOR logico fra A e Memoria |                      |      |       |                     |
|--------------------------------------|----------------------|------|-------|---------------------|
| Mnem.                                | COD. OPER<br>DEC HEX | N.B. | N.CY. | MODI INDIRIZZAMENTO |
| EOR # OPER                           | 73 49                | 2    | 2     | Immediato           |
| EOR OPER                             | 69 45                | 2    | 3     | Pagina Zero         |
| EOR OPER,X                           | 85 55                | 2    | 4     | Pagina Zero,X       |
| EOR OPER                             | 77 4D                | 3    | 4     | Assoluto            |
| EOR OPER,X                           | 93 5D                | 3    | 4*    | Assoluto,X          |
| EOR OPER,Y                           | 89 59                | 3    | 4*    | Assoluto,Y          |
| EOR (OPER,X)                         | 65 41                | 2    | 6     | (Indiretto, X)      |
| EOR (OPER),Y                         | 81 51                | 2    | 5*    | (Indiretto),Y       |

\*Aggiungere 1 ciclo se salta pagina

## CORSO DI ASSEMBLER CBM64

### INC

| Incrementa la memoria di 1 |        |                      |    |      |       |                     |
|----------------------------|--------|----------------------|----|------|-------|---------------------|
| Mnem.                      |        | COD. OPER<br>DEC HEX |    | N.B. | N.CY. | MODI INDIRIZZAMENTO |
| INC                        | OPER   | 230                  | E6 | 2    | 5     | Immediato           |
| INC                        | OPER,X | 246                  | F6 | 2    | 6     | Pagina Zero,X       |
| INC                        | OPER   | 238                  | EE | 3    | 6     | Assoluto            |
| INC                        | OPER,X | 254                  | FE | 3    | 7     | Assoluto,X          |

### INX

| Incrementa X di 1 |  |                      |    |      |       |                     |
|-------------------|--|----------------------|----|------|-------|---------------------|
| Mnem.             |  | COD. OPER<br>DEC HEX |    | N.B. | N.CY. | MODI INDIRIZZAMENTO |
| INX               |  | 232                  | E8 | 1    | 2     | Implicito           |

### INY

| Incrementa Y di 1 |  |                      |    |      |       |                     |
|-------------------|--|----------------------|----|------|-------|---------------------|
| Mnem.             |  | COD. OPER<br>DEC HEX |    | N.B. | N.CY. | MODI INDIRIZZAMENTO |
| INY               |  | 200                  | C8 | 1    | 2     | Implicito           |

# CORSO DI ASSEMBLER CBM64

## JMP

| Salto incondizionato a loc. di memoria |                      |      |       |                     |  |
|----------------------------------------|----------------------|------|-------|---------------------|--|
| Mnem.                                  | COD. OPER<br>DEC HEX | N.B. | N.CY. | MODI INDIRIZZAMENTO |  |
| JMP OPER                               | 76 4C                | 3    | 3     | Assoluto            |  |
| JMP (OPER)                             | 108 6C               | 3    | 5     | Indiretto           |  |

## JSR

| Salta ad una Subroutine |                      |      |       |                     |  |
|-------------------------|----------------------|------|-------|---------------------|--|
| Mnem.                   | COD. OPER<br>DEC HEX | N.B. | N.CY. | MODI INDIRIZZAMENTO |  |
| JSR OPER                | 32 20                | 3    | 6     | Assoluto            |  |

## LDA

| Carica in A la memoria |                      |      |       |                     |  |
|------------------------|----------------------|------|-------|---------------------|--|
| Mnem.                  | COD. OPER<br>DEC HEX | N.B. | N.CY. | MODI INDIRIZZAMENTO |  |
| LDA # OPER             | 169 A9               | 2    | 2     | Immediato           |  |
| LDA OPER               | 165 A5               | 2    | 3     | Pagina Zero         |  |
| LDA OPER,X             | 181 B5               | 2    | 4     | Pagina Zero,X       |  |
| LDA OPER               | 173 AD               | 3    | 4     | Assoluto            |  |
| LDA OPER,X             | 189 BD               | 3    | 4*    | Assoluto,X          |  |
| LDA OPER,Y             | 185 B9               | 3    | 4*    | Assoluto,Y          |  |
| LDA (OPER,X)           | 161 A1               | 2    | 6     | (Indiretto, X)      |  |
| LDA (OPER),Y           | 177 B1               | 2    | 5*    | (Indiretto),Y       |  |

\*Aggiungere 1 ciclo se salta pagina

## CORSO DI ASSEMBLER CBM64

### LDX

| Carica in X la memoria |                      |      |       |                     |  |
|------------------------|----------------------|------|-------|---------------------|--|
| Mnem.                  | COD. OPER<br>DEC HEX | N.B. | N.CY. | MODI INDIRIZZAMENTO |  |
| LDX # OPER             | 162 A2               | 2    | 2     | Immediato           |  |
| LDX OPER,Y             | 182 B6               | 2    | 4     | Pagina Zero,Y       |  |
| LDX OPER               | 166 A6               | 2    | 3     | Pagina Zero         |  |
| LDX OPER               | 174 AE               | 3    | 4     | Assoluto            |  |
| LDX OPER,Y             | 190 BE               | 3    | 4*    | Assoluto,Y          |  |

\*Aggiungere 1 ciclo se salta pagina

### LDY

| Carica in Y la memoria |                      |      |       |                     |  |
|------------------------|----------------------|------|-------|---------------------|--|
| Mnem.                  | COD. OPER<br>DEC HEX | N.B. | N.CY. | MODI INDIRIZZAMENTO |  |
| LDY # OPER             | 160 A0               | 2    | 2     | Immediato           |  |
| LDY OPER               | 166 A4               | 2    | 3     | Pagina Zero         |  |
| LDY OPER,X             | 180 B4               | 2    | 4     | Pagina Zero,X       |  |
| LDY OPER               | 172 AC               | 3    | 4     | Assoluto            |  |
| LDY OPER,X             | 188 BC               | 3    | 4*    | Assoluto,X          |  |

\*Aggiungere 1 ciclo se salta pagina

# CORSO DI ASSEMBLER CBM64

## LSR

| Esegue uno spostamento verso destra |                      |    |      |       |                     |
|-------------------------------------|----------------------|----|------|-------|---------------------|
| Mnem.                               | COD. OPER<br>DEC HEX |    | N.B. | N.CY. | MODI INDIRIZZAMENTO |
| LSR # OPER                          | 74                   | 4A | 2    | 2     | Accumulatore        |
| LSR OPER                            | 70                   | 46 | 2    | 3     | Pagina Zero         |
| LSR OPER,X                          | 56                   | 56 | 2    | 4     | Pagina Zero,X       |
| LSR OPER                            | 78                   | 4E | 3    | 4     | Assoluto            |
| LSR OPER,X                          | 94                   | 5E | 3    | 4*    | Assoluto,X          |

## NOP

| Non esegue nessuna operazione |                      |    |      |       |                     |
|-------------------------------|----------------------|----|------|-------|---------------------|
| Mnem.                         | COD. OPER<br>DEC HEX |    | N.B. | N.CY. | MODI INDIRIZZAMENTO |
| NOP                           | 234                  | EA | 1    | 2     | Implicito           |

## CORSO DI ASSEMBLER CBM64

ORA

| Esegue un OR logico fra A e la memoria |           |     |      |       |                     |
|----------------------------------------|-----------|-----|------|-------|---------------------|
| Mnem.                                  | COD. OPER |     | N.B. | N.CY. | MODI INDIRIZZAMENTO |
|                                        | DEC       | HEX |      |       |                     |
| ORA # OPER                             | 9         | 09  | 2    | 2     | Immediato           |
| ORA OPER                               | 5         | 05  | 2    | 3     | Pagina Zero         |
| ORA OPER,X                             | 21        | 15  | 2    | 4     | Pagina Zero,X       |
| ORA OPER                               | 13        | 0D  | 3    | 4     | Assoluto            |
| ORA OPER,X                             | 29        | 1D  | 3    | 4*    | Assoluto,X          |
| ORA OPER,Y                             | 25        | 19  | 3    | 4*    | Assoluto,Y          |
| ORA (OPER,X)                           | 1         | 01  | 2    | 6     | (Indiretto, X)      |
| ORA (OPER),Y                           | 17        | 11  | 2    | 5*    | (Indiretto),Y       |

\*Aggiungere 1 ciclo se salta pagina

PHA

| Memorizza l' A nello Stack |           |     |      |       |                     |
|----------------------------|-----------|-----|------|-------|---------------------|
| Mnem.                      | COD. OPER |     | N.B. | N.CY. | MODI INDIRIZZAMENTO |
|                            | DEC       | HEX |      |       |                     |
| PHA                        | 72        | 48  | 1    | 3     | Implicito           |

## CORSO DI ASSEMBLER CBM64

PHP

| Memorizza lo Status nello Stack |                      |      |       |                     |
|---------------------------------|----------------------|------|-------|---------------------|
| Mnem.                           | COD. OPER<br>DEC HEX | N.B. | N.CY. | MODI INDIRIZZAMENTO |
| PHP                             | 8 08                 | 1    | 3     | Implicito           |

PLA

| Carica in A il contenuto dello Stack |                      |      |       |                     |
|--------------------------------------|----------------------|------|-------|---------------------|
| Mnem.                                | COD. OPER<br>DEC HEX | N.B. | N.CY. | MODI INDIRIZZAMENTO |
| PLA                                  | 104 68               | 1    | 4     | Implicito           |

PLP

| Carica lo Status con lo Stack |                      |      |       |                     |
|-------------------------------|----------------------|------|-------|---------------------|
| Mnem.                         | COD. OPER<br>DEC HEX | N.B. | N.CY. | MODI INDIRIZZAMENTO |
| PLP                           | 40 28                | 1    | 4     | Implicito           |

# CORSO DI ASSEMBLER CBM64

## ROL

| Rotazione verso sinistra |        |                      |    |      |       |                     |
|--------------------------|--------|----------------------|----|------|-------|---------------------|
| Mnem.                    |        | COD. OPER<br>DEC HEX |    | N.B. | N.CY. | MODI INDIRIZZAMENTO |
| ROL                      | A      | 42                   | 2A | 1    | 2     | Accumulatore        |
| ROL                      | OPER   | 38                   | 26 | 2    | 5     | Pagina Zero         |
| ROL                      | OPER,X | 54                   | 36 | 2    | 6     | Pagina Zero,X       |
| ROL                      | OPER   | 46                   | 2E | 3    | 6     | Assoluto            |
| ROL                      | OPER,X | 62                   | 3E | 3    | 7     | Assoluto,X          |

## ROR

| Rotazione verso destra |        |                      |    |      |       |                     |
|------------------------|--------|----------------------|----|------|-------|---------------------|
| Mnem.                  |        | COD. OPER<br>DEC HEX |    | N.B. | N.CY. | MODI INDIRIZZAMENTO |
| ROR                    | A      | 106                  | 6A | 1    | 2     | Accumulatore        |
| ROR                    | OPER   | 102                  | 66 | 2    | 5     | Pagina Zero         |
| ROR                    | OPER,X | 118                  | 76 | 2    | 6     | Pagina Zero,X       |
| ROR                    | OPER   | 110                  | 6E | 3    | 6     | Assoluto            |
| ROR                    | OPER,X | 126                  | 7E | 3    | 7     | Assoluto,X          |

## RTI

| Ritorno da INTERRUPT |  |                      |    |      |       |                     |
|----------------------|--|----------------------|----|------|-------|---------------------|
| Mnem.                |  | COD. OPER<br>DEC HEX |    | N.B. | N.CY. | MODI INDIRIZZAMENTO |
| RTI                  |  | 64                   | 40 | 1    | 6     | Implicito           |

# CORSO DI ASSEMBLER CBM64

## RTS

| Ritorno da Subroutine |                      |      |       |                     |
|-----------------------|----------------------|------|-------|---------------------|
| Mnem.                 | COD. OPER<br>DEC HEX | N.B. | N.CY. | MODI INDIRIZZAMENTO |
| RTS                   | 96 60                | 1    | 6     | Implicito           |

## SBC

| Sottrae A da memoria con Carry. |                      |      |       |                     |
|---------------------------------|----------------------|------|-------|---------------------|
| Mnem.                           | COD. OPER<br>DEC HEX | N.B. | N.CY. | MODI INDIRIZZAMENTO |
| SBC # OPER                      | 233 E9               | 2    | 2     | Immediato           |
| SBC OPER                        | 229 E5               | 2    | 3     | Pagina Zero         |
| SBC OPER,X                      | 245 F5               | 2    | 4     | Pagina Zero,X       |
| SBC OPER                        | 237 ED               | 3    | 4     | Assoluto            |
| SBC OPER,X                      | 253 FD               | 3    | 4*    | Assoluto,X          |
| SBC OPER,Y                      | 249 F9               | 3    | 4*    | Assoluto,Y          |
| SBC (OPER,X)                    | 225 E1               | 2    | 6     | (Indiretto, X)      |
| SBC (OPER),Y                    | 241 F1               | 2    | 5*    | (Indiretto),Y       |

\*Aggiungere 1 ciclo se salta pagina

# CORSO DI ASSEMBLER CBM64

## SEC

| Pone a 1 il bit di Carry |                      |      |       |                     |
|--------------------------|----------------------|------|-------|---------------------|
| Mnem.                    | COD. OPER<br>DEC HEX | N.B. | N.CY. | MODI INDIRIZZAMENTO |
| SEC                      | 56 38                | 1    | 2     | Implicito           |

## SED

| Pone a 1 il Flag D |                      |      |       |                     |
|--------------------|----------------------|------|-------|---------------------|
| Mnem.              | COD. OPER<br>DEC HEX | N.B. | N.CY. | MODI INDIRIZZAMENTO |
| SED                | 248 F8               | 1    | 2     | Implicito           |

## SEI

| Mette a 1 il bit di interrupt |                      |      |       |                     |
|-------------------------------|----------------------|------|-------|---------------------|
| Mnem.                         | COD. OPER<br>DEC HEX | N.B. | N.CY. | MODI INDIRIZZAMENTO |
| SEI                           | 120 78               | 1    | 2     | Implicito           |

# CORSO DI ASSEMBLER CBM64

## STA

| Scri ve il contenuto di A in Memoria |          |                      |    |      |       |                     |
|--------------------------------------|----------|----------------------|----|------|-------|---------------------|
| Mnem.                                |          | COD. OPER<br>DEC HEX |    | N.B. | N.CY. | MODI INDIRIZZAMENTO |
| STA                                  | OPER     | 133                  | 85 | 2    | 3     | Pagina Zero         |
| STA                                  | OPER,X   | 149                  | 95 | 2    | 4     | Pagina Zero,X       |
| STA                                  | OPER     | 141                  | 8D | 3    | 4     | Assoluto            |
| STA                                  | OPER,X   | 157                  | 9D | 3    | 4*    | Assoluto,X          |
| STA                                  | OPER,Y   | 153                  | 99 | 3    | 4*    | Assoluto,Y          |
| STA                                  | (OPER,X) | 129                  | 81 | 2    | 6     | (Indiretto, X)      |
| STA                                  | (OPER),Y | 145                  | 91 | 2    | 5*    | (Indiretto),Y       |

## STX

| Scrive X in memoria |        |                      |    |      |       |                     |
|---------------------|--------|----------------------|----|------|-------|---------------------|
| Mnem.               |        | COD. OPER<br>DEC HEX |    | N.B. | N.CY. | MODI INDIRIZZAMENTO |
| STX                 | OPER   | 134                  | 86 | 2    | 3     | Pagina Zero         |
| STX                 | OPER,Y | 150                  | 96 | 2    | 4     | Pagina Zero,X       |
| STX                 | OPER   | 142                  | 8E | 3    | 4     | Assoluto            |

## STY

| Scrive Y in Memoria |        |                      |    |      |       |                     |
|---------------------|--------|----------------------|----|------|-------|---------------------|
| Mnem.               |        | COD. OPER<br>DEC HEX |    | N.B. | N.CY. | MODI INDIRIZZAMENTO |
| STY                 | OPER   | 132                  | 84 | 2    | 3     | Pagina Zero         |
| STY                 | OPER,X | 148                  | 94 | 2    | 4     | Pagina Zero,X       |
| STY                 | OPER   | 140                  | 8C | 3    | 4     | Assoluto            |

# CORSO DI ASSEMBLER CBM64

TAX

| Trasferisce A in X |                      |      |       |                     |
|--------------------|----------------------|------|-------|---------------------|
| Mnem.              | COD. OPER<br>DEC HEX | N.B. | N.CY. | MODI INDIRIZZAMENTO |
| TAX                | 170 AA               | 1    | 2     | Implicito           |

TAY

| Trasferisce A in Y |                      |      |       |                     |
|--------------------|----------------------|------|-------|---------------------|
| Mnem.              | COD. OPER<br>DEC HEX | N.B. | N.CY. | MODI INDIRIZZAMENTO |
| TAY                | 168 A8               | 1    | 2     | Implicito           |

TSX

| Trasferisce lo Stack in X |                      |      |       |                     |
|---------------------------|----------------------|------|-------|---------------------|
| Mnem.                     | COD. OPER<br>DEC HEX | N.B. | N.CY. | MODI INDIRIZZAMENTO |
| TSX                       | 186 BA               | 1    | 2     | Implicito           |

# CORSO DI ASSEMBLER CBM64

TXA

| Trasferisce X in A |                      |      |       |                     |
|--------------------|----------------------|------|-------|---------------------|
| Mnem.              | COD. OPER<br>DEC HEX | N.B. | N.CY. | MODI INDIRIZZAMENTO |
| TXA                | 138 8A               | 1    | 2     | Implicito           |

TXS

| Copia X nello Stack |                      |      |       |                     |
|---------------------|----------------------|------|-------|---------------------|
| Mnem.               | COD. OPER<br>DEC HEX | N.B. | N.CY. | MODI INDIRIZZAMENTO |
| TXS                 | 154 9A               | 1    | 2     | Implicito           |

TYA

| Copia Y in A |                      |      |       |                     |
|--------------|----------------------|------|-------|---------------------|
| Mnem.        | COD. OPER<br>DEC HEX | N.B. | N.CY. | MODI INDIRIZZAMENTO |
| TYA          | 152 98               | 1    | 2     | Implicito           |

# CORSO DI ASSEMBLER CBM64

| COD. | CARATTERE                       | COD. | CARATT. | COD. | CARATTERE    |
|------|---------------------------------|------|---------|------|--------------|
| 00   | NUL : Null                      | 30   | 0       | 60   | -            |
| 01   | SOH : Start of Heading          | 31   | 1       | 61   | a            |
| 02   | STX : Start of Text             | 32   | 2       | 62   | b            |
| 03   | ETX : End of Text               | 33   | 3       | 63   | c            |
| 04   | EOT : End of Transmission       | 34   | 4       | 64   | d            |
| 05   | ENQ : Enquiry                   | 35   | 5       | 65   | e            |
| 06   | ACK : Acknowledge               | 36   | 6       | 66   | f            |
| 07   | BEL : Bell                      | 37   | 7       | 67   | g            |
| 08   | BS : Backspace                  | 38   | 8       | 68   | h            |
| 09   | HT : Horizontal Tabulation      | 39   | 9       | 69   | i            |
| 0A   | LF : Line Feed                  | 3A   | :       | 6A   | j            |
| 0B   | VT : Vertical Tabulation        | 3B   | :       | 6B   | k            |
| 0C   | FF : Form Fedd                  | 3C   | <       | 6C   | l            |
| 0D   | CR : Carriage Return            | 3D   | =       | 6D   | m            |
| 0E   | SO : Shift out                  | 3E   | >       | 6E   | n            |
| 0F   | SI : Shift in                   | 3F   | ?       | 6F   | o            |
| 10   | DLE : Data link Escape          | 40   | @       | 70   | p            |
| 11   | DC1 : Device Control 1          | 41   | A       | 71   | q            |
| 12   | DC2 : - 2                       | 42   | B       | 72   | r            |
| 13   | DC3 : - 3                       | 43   | C       | 73   | s            |
| 14   | DC4 : - 4                       | 44   | D       | 74   | t            |
| 15   | NAK : Negative Acknowledge      | 45   | E       | 75   | u            |
| 16   | SYN : Synchronous Idle          | 46   | F       | 76   | v            |
| 17   | ETB : End of Transmission Block | 47   | G       | 77   | w            |
| 18   | CAN : Cancel                    | 48   | H       | 78   | x            |
| 19   | EM : End of Medium              | 49   | I       | 79   | y            |
| 1A   | SUB : Substitute                | 4A   | J       | 7A   | z            |
| 1B   | ESC : Escape                    | 4B   | K       | 7B   | }            |
| 1C   | FS : File Separator             | 4C   | L       | 7C   | _            |
| 1D   | GS : Group Separator            | 4D   | M       | 7D   | }            |
| 1E   | RS : Record Separator           | 4E   | N       | 7E   | ~            |
| 1F   | US : Unit Separotor             | 4F   | O       | 7F   | DEL : Delete |
| 20   | SP : Space                      | 50   | P       |      |              |
| 21   | !                               | 51   | Q       |      |              |
| 22   | -                               | 52   | R       |      |              |
| 23   | #                               | 53   | S       |      |              |
| 24   | \$                              | 54   | T       |      |              |
| 25   | %                               | 55   | U       |      |              |
| 26   | &                               | 56   | V       |      |              |
| 27   | .                               | 57   | W       |      |              |
| 28   | (                               | 58   | X       |      |              |
| 29   | )                               | 59   | Y       |      |              |
| 2A   | *                               | 5A   | Z       |      |              |
| 2B   | +                               | 5B   | [       |      |              |
| 2C   | ,                               | 5C   | \       |      |              |
| 2D   | -                               | 5D   | ]       |      |              |
| 2E   | .                               | 5E   | ^       |      |              |
| 2F   | /                               | 5F   | _       |      |              |

## CORSO DI ASSEMBLER CBM64

### LEZIONE QUARTA

|                         |    |
|-------------------------|----|
| Istruzioni di confronto | 55 |
| I Flags del 6510        | 59 |
| Il Flag N               | 60 |

### LEZIONE QUINTA

|                               |    |
|-------------------------------|----|
| Le altre istruzioni           | 63 |
| Temporizzazione dei programmi | 68 |

### LEZIONE SESTA

|                                 |    |
|---------------------------------|----|
| Modi di indirizzamento          | 75 |
| Indirizzamento immediato        | 76 |
| Indirizzamento implicito        | 78 |
| Indirizzamento assoluto         | 79 |
| Indirizzamento pagina zero      | 79 |
| Indirizzamento indicizzato ass. | 81 |
| Indirizzamento relativo         | 81 |
| Indirizzamento indiretto        | 82 |
| Uso del registro X              | 82 |
| Uso del registro Y              | 84 |
| Indirizzamento indiretto ass.   | 86 |

### LEZIONE SETTIMA

|                                 |     |
|---------------------------------|-----|
| Operazioni matematiche          | 87  |
| Operazioni in doppia precisione | 87  |
| Input in esadecimale            | 95  |
| La moltiplicazione              | 98  |
| La divisione                    | 101 |

### LEZIONE OTTAVA

|                         |     |
|-------------------------|-----|
| Codice decimale binario | 103 |
|-------------------------|-----|

## CORSO DI ASSEMBLER CBM64

|                                  |     |
|----------------------------------|-----|
| Operatori logici                 | 104 |
| Tavola della verita' AND         | 107 |
| Tavola della verita' OR          | 111 |
| Tavola della verita' EOR         | 113 |
| Altre forme di manipolazione BIT | 114 |

### LEZIONE NONA

|                           |     |
|---------------------------|-----|
| Moltiplicazione binaria   | 121 |
| Moltiplicazione a 8 bit   | 123 |
| Divisione binaria a 8 bit | 127 |

### LEZIONE DECIMA

|                              |     |
|------------------------------|-----|
| Le labels                    | 129 |
| Memory labels                | 131 |
| Il Monitor                   | 134 |
| Altre informazioni sul BASIC | 141 |

### LEZIONE UNDICESIMA

|                          |     |
|--------------------------|-----|
| Collegamento col Basic   | 143 |
| Protezione dei programmi | 147 |

### LEZIONE DODICESIMA

|                    |     |
|--------------------|-----|
| I colori nel CBM64 | 151 |
| Tavola dei colori  | 154 |

### LEZIONE TREDICESIMA

|                      |     |
|----------------------|-----|
| Le routines del S.O. | 155 |
|----------------------|-----|

### LEZIONE QUATTORDICESIMA

|                           |     |
|---------------------------|-----|
| Routines di sistema CHRIN | 163 |
|---------------------------|-----|

## CORSO DI ASSEMBLER CBM64

|                         |     |
|-------------------------|-----|
| Lo Stack                | 168 |
| Schema di funzionamento | 174 |

### LEZIONE QUINDICESIMA

|                                |     |
|--------------------------------|-----|
| Interrupts, numeri e variabili | 175 |
| Numeri relativi                | 181 |
| Gli overflow                   | 184 |
| Visualizzazione dei numeri     | 186 |

### LEZIONE SEDICESIMA

|                             |     |
|-----------------------------|-----|
| Numeri in virgola mobile    | 189 |
| Il comando USR              | 191 |
| Subrout. per virgola mobile | 195 |
| Addizione                   | 202 |
| Sottrazione                 | 203 |
| Divisione                   | 204 |
| Potenze                     | 204 |
| Logaritmi                   | 205 |
| Uso della stampante         | 206 |

|                          |     |
|--------------------------|-----|
| SOLUZIONE DEGLI ESERCIZI | 209 |
|--------------------------|-----|

### APPENDICI

|                        |  |
|------------------------|--|
| Le routines Kernal     |  |
| Tavole istruzioni 6510 |  |

#### **CORSO DI ASSEMBLER CBM64**

Una delle cose che scoraggiano maggiormente il normale utente del computer ad andare oltre il piu o meno completo apprendimento del Basic e il numero delle parole e piu in generale dei termini da imparare.

Questo corso si propone di insegnare ad un normale utente che non abbia necessariamente conoscenze di elettronica ne di programmazione, ad usare il linguaggio Assembler. E' infatti questa la strada per scrivere velocissime routines o interi programmi con un nuovo linguaggio e un nuovo sistema che danno una dimensione completamente diversa al lavoro sul computer CBM64.

La struttura, completamente AUTODIDATTICA e articolata in:

- 17 Lezioni
- 75 Pagine di appendice
- 72 Programmi ampiamente commentati
- 23 Esercizi con le relative soluzioni
- 304 Pagine in totale con esempi e tavole

Nastro contenente un potente ASSEMBLER (AUTO, DELETE, FIND, FORMAT, TABLE, ASSEMBLE, DISASSEMBLE, ecc) con in piu' la particolarita' di essere utilizzato in maniera molto simile al Basic (numerazione delle linee, possibilita' dei REM, listabilita', ecc.). Inoltre, sempre su nastro, un MONITOR in Linguaggio Macchina.

**L. 38.000 I.V.A. compresa**

(37255)

**Completo di cassetta**

Via Marconi, 9/A - 52025 MONTEVARCHI (AR)

Tel. (055) 98.02.42 - 98.25.13